

# Introduction to the summer school mini-app code

Ben Cumming

Swiss National Supercomputing Center (CSCS)



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

CSCS

Swiss National Supercomputing Centre



# Overview

- What is a mini-app?
- An overview of our mini-app
- First look at the code.
- Run the code and visualize output



# HPC Mini-Apps

- Full HPC applications have complicated behavior
  - difficult to model or understand performance behavior
- Mini applications (mini-apps) are smaller codes that aim to characterize larger applications
  - typically a few hundred to a few thousand lines of code
- Are simpler to test and understand than full applications
- Used to test different hardware and programming languages
- Good for learning new techniques!



# Our Mini-App

- Throughout the summer school we will be using a mini-app to reinforce the lessons
  - During talks there will be small programming exercises to test out what you learn
  - Then you will get the opportunity to apply the techniques to the mini-app
- We will start with a serial version that has no parallel optimizations
  - By the end of the course we will have several different versions, one for each technique

# The Application

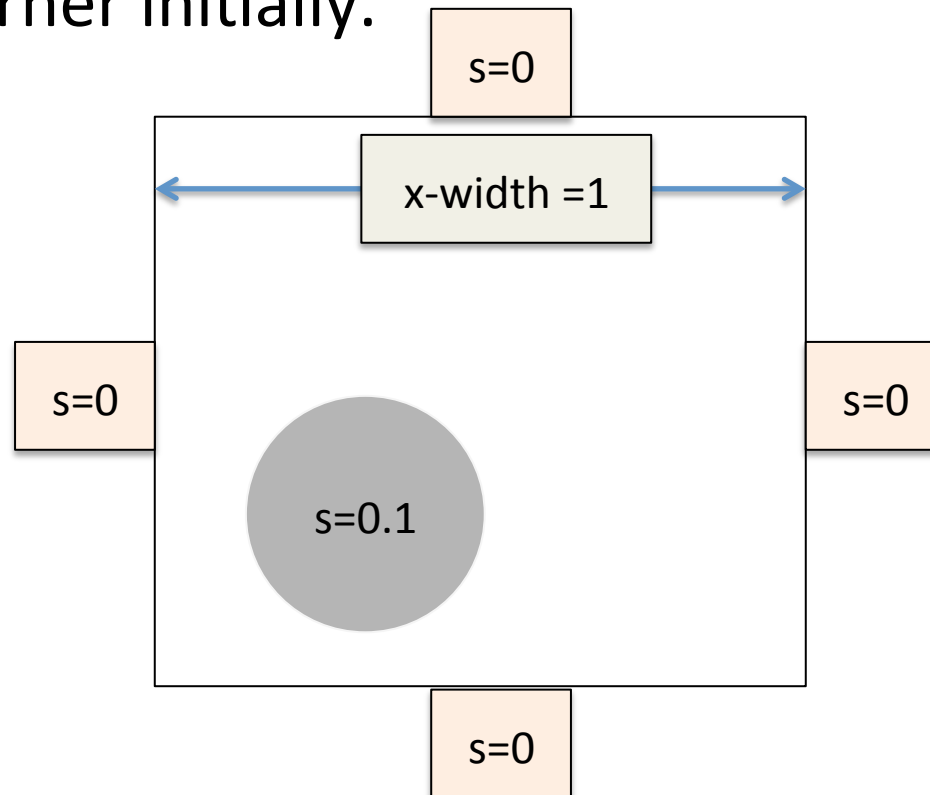
- The code solves a **reaction diffusion** equation known as **Fischer's Equation**

$$\frac{\partial s}{\partial t} = D \overbrace{\left( \frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2} \right)}^{\text{diffusion}} + \overbrace{R s (1 - s)}^{\text{reaction/growth}}$$

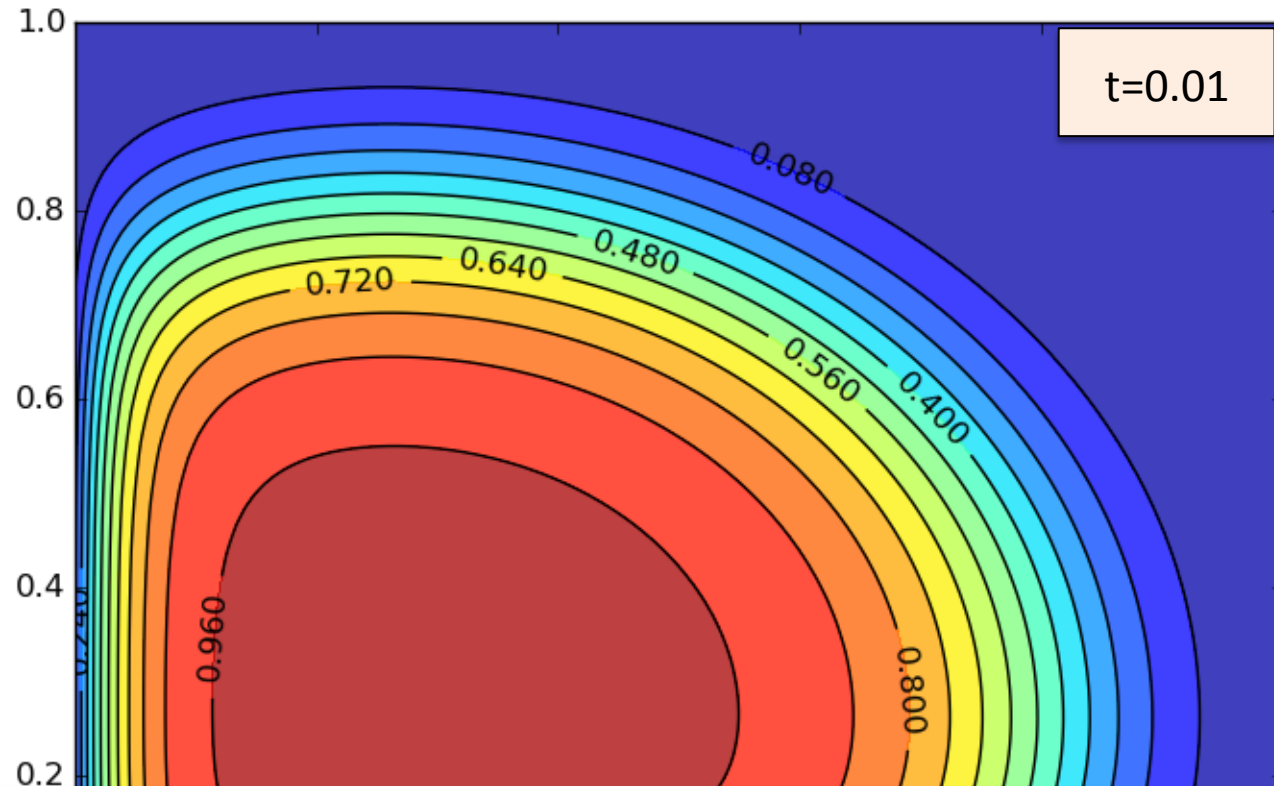
- Used to simulate travelling waves and simple population dynamics
  - The species **s** diffuses
  - And the population grows to a maximum of **s=1**

# Initial and boundary conditions

- The domain is rectangular, with fixed value of  $s=0$  on each boundary, and a circular region of  $s=0.1$  in the lower left corner initially.



# Time Evolution of Solution



- For most cases we will run the solution until  $t=0.01$ .
- Long enough for something interesting to happen
  - Will clearly show if there is a problem

# Numerical Solution

- The rectangular domain is discretized with a grid of dimension  $n_x * n_y$  points
- A **finite volume** discretization and **method of lines** gives the following ordinary differential equation for each grid point

$$\frac{ds_{ij}}{dt} = \frac{D}{\Delta x^2} [-4s_{ij} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1}] + Rs_{ij}(1 - s_{ij})$$

- Which we can express as the following nonlinear problem...

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





# Numerical Solution

- One nonlinear equation for each grid point
  - together they form a system of  $N=n_x*n_y$  equations
  - Solve with Newton's method
- Each iteration of Newton's method has to solve a linear system
  - Solve with matrix-free Conjugate Gradient solver
- Solve the nonlinear system at each time step
  - This requires in the order of between 5-10 conjugate gradient iterations

- Don't worry if you don't understand it all!
- We don't need a deep understanding of the mathematics or domain problem to optimize the code
  - I often work on codes with little domain knowledge
- The mini-app has a handful of kernels that can be parallelized
  - And care was taken when designing it to make parallelization as easy as possible
- So let's look a little closer at each part of the code




# The Code

- There are two versions of the code
  - C++
  - Fortran90
- Both codes have the same structure
  - Pick whichever version you are most comfortable with
- The code could be faster
  - Fortran version is faster than the C++ version
  - We avoided aggressive optimization to make it as easy as possible to understand.
  - Neither are they fine examples of design, which would get in the way of understanding.
- Tested with both Cray, GNU, Intel and PGI compilers



# Code Walkthrough

- There are three modules of interest
  - [main.f90/main.cpp](#) : initialization and main time stepping loop
  - [linalg.f90/linalg.cpp](#) : the BLAS level 1 (vector-vector) kernels and conjugate gradient solver
  - [operators.f90/operators.cpp](#) : the stencil operator for the finite volume discretization



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

# Linear algebra: linalg.f90/cpp

- This file defines simple kernels for operating on 1D vectors, including
  - dot product :  $\mathbf{x} \bullet \mathbf{y}$  : `ss_dot`
  - linear combination :  $\mathbf{z} = \alpha * \mathbf{x} + \beta * \mathbf{y}$  : `ss_lcomb`
- The kernels of interest start with `ss_XXXXX`
  - `ss` == summer school
- For each parallelization approach that we will see (OpenMP, MPI, CUDA, ... etc), each of these kernels will have to be considered.



# Stencil operator: operator.f90/cpp

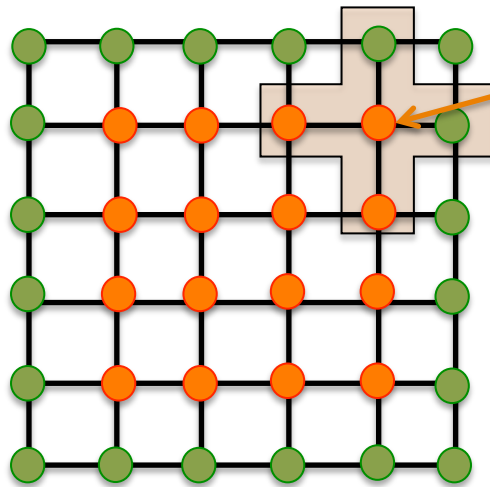
- This file has a function/subroutine that defines the stencil operator

interior points

```
for j=2:ydim-1
  for i=2:xdim-1
    
$$f_{ij} = [- (4 + \alpha) s_{ij} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{ij} (1 - s_{ij})]^{k+1} + \alpha s_{ij}^k = 0$$

  end
end
```

# Stencil: Interior Grid Points



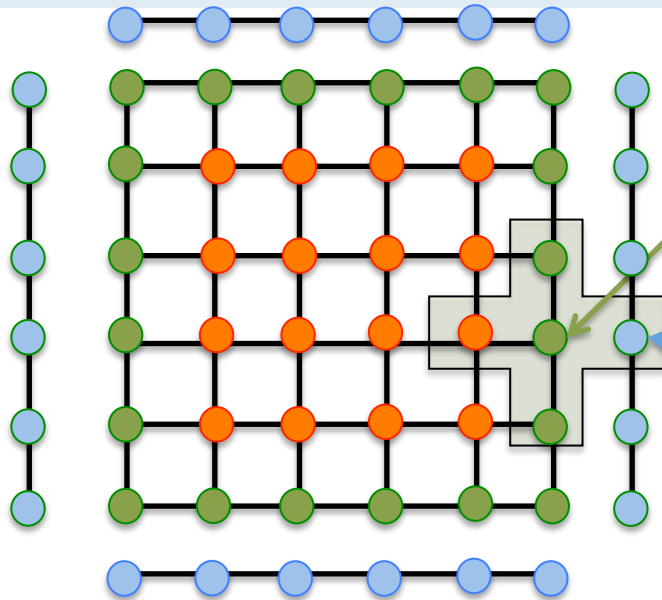
interior points have all  
neighbours available

interior points

```
for j=2:ydim-1
  for i=2:xdim-1
    
$$f_{ij} = [- (4 + \alpha) s_{ij} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{ij} (1 - s_{ij})]^{k+1} + \alpha s_{ij}^k = 0$$

  end
end
```

# Stencil: Boundary Grid Points



boundary points are missing  
1 or 2 neighbours

create 4 halo buffers, that  
hold "ghost" buffers  
bndN, bndE, bndS, bndW

east boundary

```
i=xdim
for j=2:ydim-1
```

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

```
end
```



# Testing the code

- Get the code, by checking it out from github

```
> git clone https://github.com/eth-cscs/SummerSchool2016.git
> cd SummerSchool2016/miniapp/serial
> ls
cxx fortran
> cd cxx
```

I choose the C++ version here

- Compile and run

```
> make
> srun -n1 ./main 128 128 100 0.01 yes
```



# Testing continued...

- Compile

```
> make
```

- Run interactively (use salloc beforehand)

```
> srun ./main 128 128 100 0.01
```

- the grid is 128 x 128 grid points
- take 100 time steps
- run simulation for  $t=0.01$

It is possible to choose parameters that will make the simulation fail to converge! The code should tell you gracefully that it was unable to converge.

- Or run batch job

```
> sbatch job.daint  
... when job is finished ...  
> cat job.out
```

increasing the spatial resolution may require increasing the number of time steps

# Exercise

- Compile using the cray programming environment.
  - get time to solution for, note time to solution and total conjugate iterations in each case
    - 128 128 100 0.01
    - 256 256 200 0.01
    - 1024 1024 ??? 0.01
- Recompile using the GNU programming environment
  - this will require make clean to remove previous build:

```
> module swap PrgEnv-cray PrgEnv-gnu  
> make clean  
> make
```

- rerun tests above and compare time to solution and the number of conjugate iterations for each case



# Output

```
=====
version :
mesh
time :
```

The number of conjugate gradient iterations, which should always be constant for a given mesh size and time parameters. Can be used to check that changes to the code are still getting the correct result. **There will be small variations due to the imprecise nature of floating point operations.**

```
=====
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 7.21951e-07
step 100 required 12 iterations for residual 7.21951e-07
```

time to solution

```
-----
simulation took 1.58408 seconds
8127 conjugate gradient iterations, at rate of 5130.43 iters/second
920 newton iterations
-----
```

Goodbye!

best way to compare different implementations



# Visualize the answer

- The application generates two data files with the final solution: `output.bin` and `output.bov`
- There is a Python script that will show a contour plot of the solution

```
> ls output.*
output.bin output.bov
> module load python/2.7.6
> python plotting.py
> display output.png
```

requires X-windowing  
make sure you connect with "ssh -X"

# Exercise

- Visualize the output from the previous exercise
  - now is a good time to see if X-windows is working!
- If that worked properly, try visualizing output from different final times
  - `srun -n1 ./main 128 128 100 0.0025`
  - `srun -n1 ./main 128 128 100 0.005`
  - `srun -n1 ./main 128 128 100 0.01`

# Questions?

- You will become more familiar with the mini-app code over the summer-school.

