Introduction to the summer school mini-app code

Ben Cumming

Swiss National Supercomputing Center (CSCS)

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 too 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

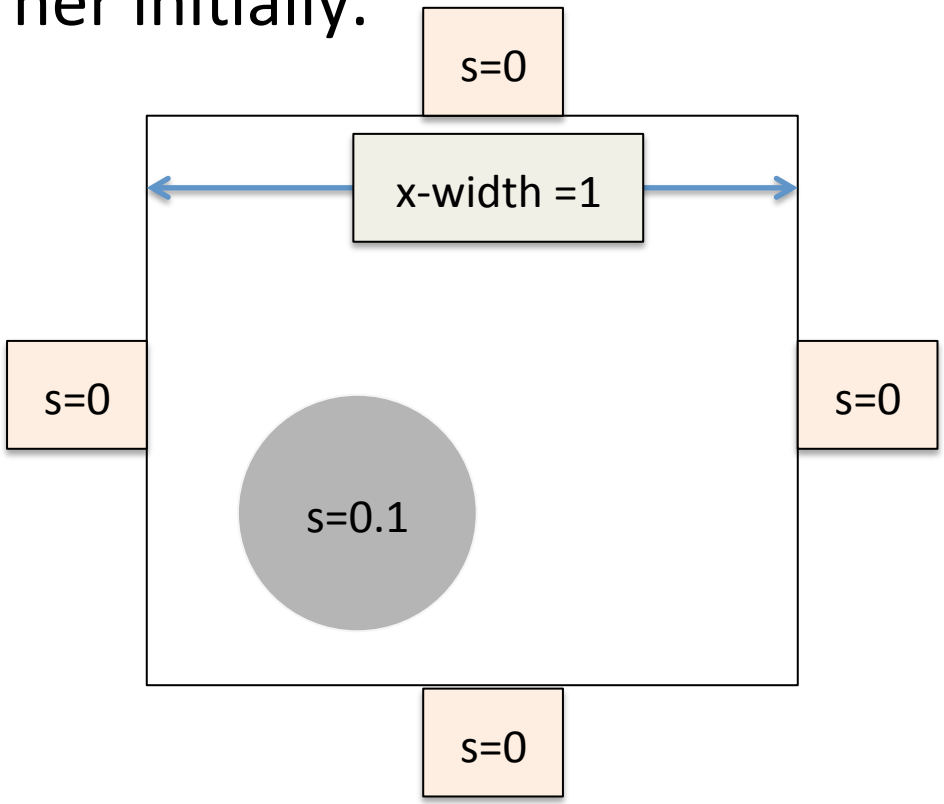
- The code solves a reaction diffusion equation known as Fischer's Equation

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

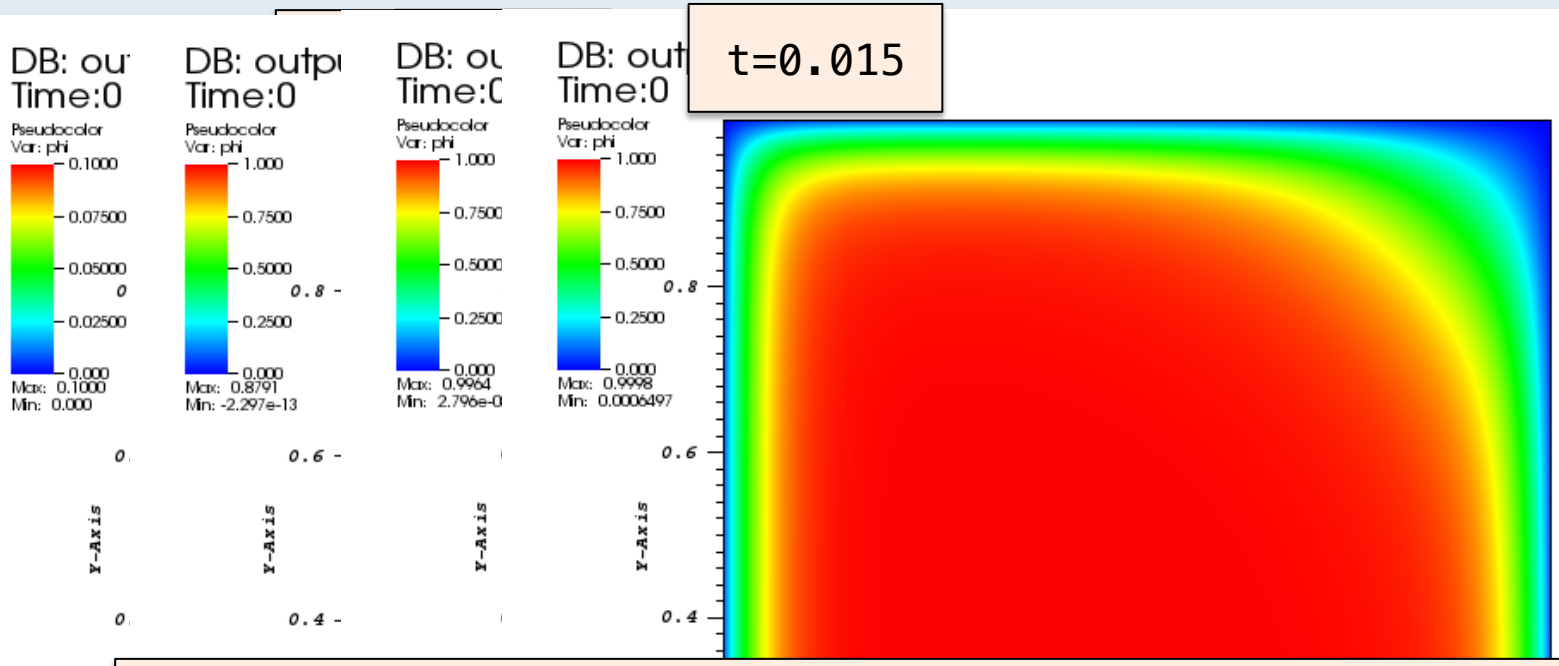
- Used to simulate travelling waves and simple population dynamics
  - The species s diffusions from central starting point
  - And the population grows to a maximum of s=1

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 nx*ny points
- A finite volume discretization and method of lines gives the follow ordinary differential equation for each grid point

$$\frac{\mathrm{d}s_{ij}}{\mathrm{d}t} = \frac{D}{\Delta x^2}\left[-4s_{ij} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1}\right] + Rs_{ij}\left(1 - s_{ij}\right)$$

- Which we can express as the following nonlinear problem…

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

Numerical Solution

- We have one nonlinear equation for each grid point
  - together they form a system of N=nx*ny 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
- We solve one nonlinear system at each time step
  - This requires in the order of between 5-10 conjugate gradient iterations

I'm not a Mathematician!

- Don't worry!
- We don't need a deep understanding of the mathematics or domain problem to optimize the code
- 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++ : Nothing fancy, just C with some C++ features.
  - Fortran90
- Both codes have the same structure
  - And have very similar time to solution
  - Pick whichever version you are most comfortable with
- The code could be faster (It is fairly fast, not very fast)
  - 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.
- Have been tested with both Cray and GNU 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 : **x•y** : ss_dot
  - linear combination : **z**=*alpha****x** + *beta****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

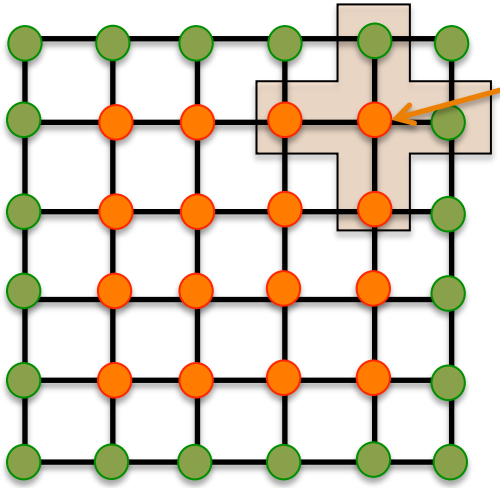
interior points

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

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

```
  end
end
```


Stencil: Interior Grid Points



interior points have all neighbours available

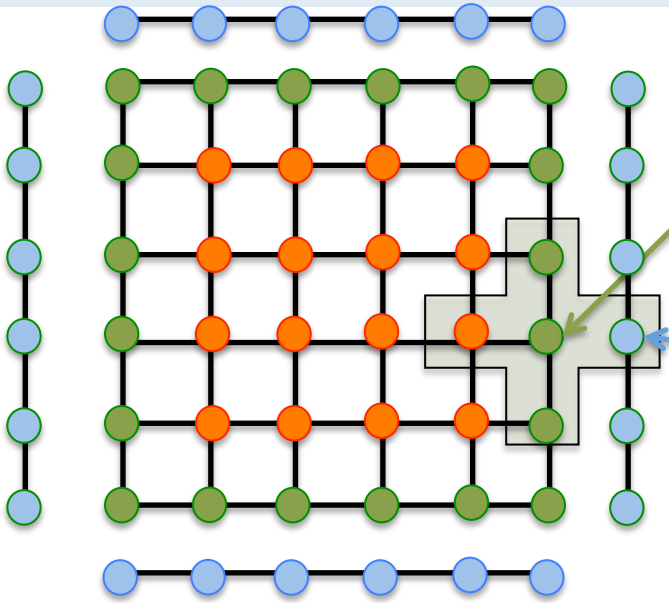
interior points

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

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

```
  end
end
```

Stencil: Boundary Grid Points



boundary points are missing
1 or 2 nieghbours

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

east boundary

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

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

```
end
```

Testing the code

- Get the code, by checking it out from github

```
> svn checkout https://github.com/fomics/SummerSchool2014.git
> cd SummerSchool2014/trunk/miniapp/serial
> ls
cxx fortran
> cd cxx
```

I choose the C++ version here

- Compile and run

```
> make
> aprun ./main 128 128 100 0.01 yes
```

Testing continued...

- Compile

```
> make
```

- Run interactively (use salloc beforehand)

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

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

- Or run batch job

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

```
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.


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
    - 256 256 100 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    :
=====================================================================
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 res
step 100 required 12 iterations for re
---------------------------------------------------------------------

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

Goodbye!
```

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.

time to solution

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 script for automagically visualizes for you

```
> ls output.*
output.bin output.bov
> ./make_viz.sh
...
======== running visit to generate image ========
VisIt: Message - Saved phi_image.0001.png
======== drawing phi_image.0001.png ========
```

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

  - configure programming environment
  - call visualization code Visit to generate the image
  - open ImageMagick to render the image

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
  - aprun ./main 128 128 100 0.0025
  - aprun ./main 128 128 100 0.005
  - aprun ./main 128 128 100 0.01

Questions?

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