Università della Svizzera italiana
Institute of Computational Science ICS

HPC Lab, Fall Semester 2020
Lecturer: Prof. O. Schenk
Assistants: A. Eftekhari, R. Janalik, J. Kardoš, D. Pasadakis, T. Holt

Project 4
Parallel Space Solution of a Nonlinear PDE using OpenMP
Due date: 30 October 2020, 12pm (midnight)

The Fisher's equation as an example of a reaction-diffusion PDE

The simple OpenMP exercises such as the π-computation or the parallel Mandelbrot set are good examples for the basic understanding of OpenMP constructs, but due to their simplicity they are far away from practical applications. In the HPC Lab for CSE we are going to use a parallel PDE mini-application that solves something more sophisticated, but where the code is nevertheless still relatively short and easy to understand. Our mini-application will solve a prototype of a reaction-diffusion equation (1). It is the Fisher's Equation that can be used to simulate travelling waves and simple population dynamics, and is given by

$$\frac{\partial s}{\partial t} = D(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}) + Rs(1-s) \quad \text{in } \Omega, \tag{1}$$

where $s := s(x, y, t)$, D is the diffusion constant and R is the reaction constant. The left hand side represents the rate of change of s with time. On the right hand side, the first term describes the diffusion of s in space and the second term describes the reaction or growth of the wave or population. We set the domain to be the unit square, i.e. $\Omega = (0, 1)^2$, and prescribe Dirichlet boundary conditions on the whole boundary $\partial\Omega$ with a fixed constant value of zero. The initial conditions s^{init} are set to zero over the entire domain, except for a circular region in the bottom left corner that is initialized to the value 0.1. The domain Ω is discretized using a uniform grid with $(n+2)\times(n+2)$ points, where a grid point is indicated by $x_{i,j}$, for $i, j \in \{0, 1, ..., n+1\}$, and where $s^k_{i,j}$ is the approximation of s at the grid-point $x_{i,j}$ at time-step k, see Figure 1.
We use a second-order finite difference discretization to approximate the spatial derivatives of s for all inner grid points for a fixed time t,

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

for all $(i, j) \in \{1, \ldots, n\}$, and where $\Delta x = 1/(n+1)$, for Δx sufficiently small. In order to approximate the time derivative, we use a first-order finite difference scheme for each grid point $x_{i,j}$, which at time-step k gives

$$(\frac{\partial s}{\partial t})^k_{i,j} \approx \frac{1}{\Delta t}(s^k_{i,j} - s^{k-1}_{i,j}), \tag{3}$$

where Δt is the time-step size. Putting together the components above, we obtain the following discretization of Equation (1):

$$\frac{1}{\Delta t}(s^k_{i,j} - s^{k-1}_{i,j}) = \frac{D}{\Delta x^2}(-4s^k_{i,j} + s^k_{i-1,j} + s^k_{i+1,j} + s^k_{i,j-1} + s^k_{i,j+1}) + Rs^k_{i,j}(1 - s^k_{i,j}), \tag{4}$$

that we will attempt to solve in order to obtain an approximate solution for s. We can reformulate (4) as

$$f^k_{i,j} := [-(4+\alpha)s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j}(1 - s_{i,j})]^k + \alpha s^{k-1}_{i,j} = 0 \tag{5}$$

for each tuple (i, j) and time-step k with $\alpha := \Delta x^2/(D\Delta t)$ and $\beta := R\Delta x^2/D$. At time-step $k = 1$, we initialize $s^{k-1}_{i,j} = s^0_{i,j} := s^{\text{init}}(x_{i,j})$, and we look for approximate values $s^k_{i,j}$ that fulfill Equation (5). For all (i, j) at a fixed k,

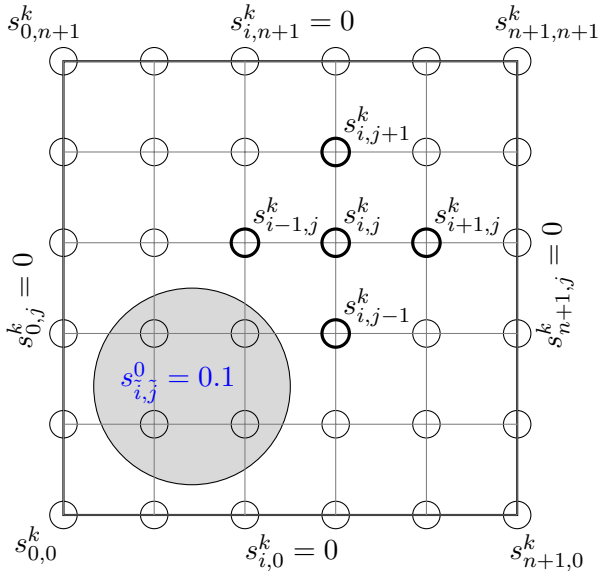


Figure 1: Discretization of the domain Ω.

we obtain a system of $N = n^2$ equations. We can see that each equation is quadratic in $s^k_{i,j}$, and therefore nonlinear which makes the problem more complicated to solve. We tackle this problem using Newton's method with which we iteratively try to find better approximations of the solution of Equation (5). In order to formulate the Newton iteration we introduce the following notation: Let $\mathbf{s}^k = [s^k_{1,1}, \ldots, s^k_{n,1}, s^k_{1,2}, \ldots, s^k_{n,n}]^T \in \mathbb{R}^N$ be a vectorized version of the approximate solution at time-step k. Then, we can consider the set of equations $f^k_{i,j}$ as functions depending on $\mathbf{s}^k$ and define $\mathbf{f}(\mathbf{s}^k) := [f^k_{1,1}, \ldots, f^k_{n,1}, f^k_{1,2}, \ldots, f^k_{n,n}]^T \in \mathbb{R}^N$. For Newton's method, we then have

$$\mathbf{y}^{l+1} = \mathbf{y}^l - [\mathbf{J}_\mathbf{f}(\mathbf{y}^l)]^{-1}\mathbf{f}(\mathbf{y}^l), \tag{6}$$

where $\mathbf{J}_\mathbf{f}(\mathbf{y}^l) \in \mathbb{R}^{N\times N}$ is the Jacobian of $\mathbf{f}$. We start with the initial guess $\mathbf{y}^0 := \mathbf{s}^{k-1}$. However, for each iteration the inverse of the Jacobian $[\mathbf{J}_\mathbf{f}(\mathbf{y}^l)]^{-1}$ is not readily available. We do not compute it directly but instead use a matrix-free Conjugate Gradient solver that solves the following linear system of equations for $\delta\mathbf{y}^{l+1}$:

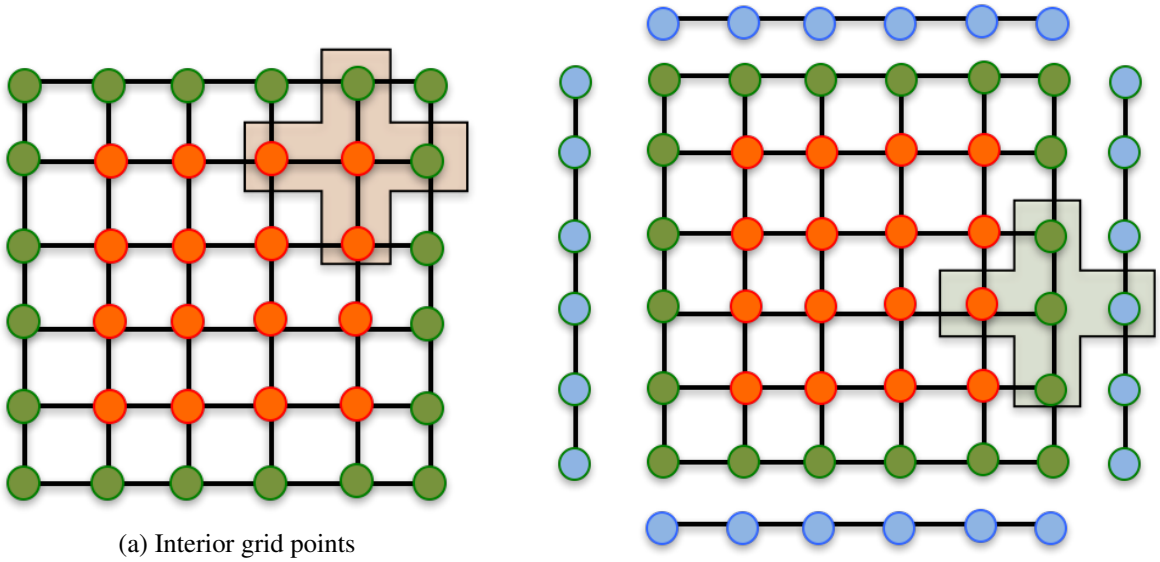
$$[\mathbf{J}_\mathbf{f}(\mathbf{y}^l)]\delta\mathbf{y}^{l+1} = \mathbf{f}(\mathbf{y}^l) \tag{7}$$

and for which it follows that $\mathbf{y}^{l+1} = \mathbf{y}^l - \delta\mathbf{y}^{l+1}$. We iterate over l in Equation (6) till a stopping criterion is reached and we obtain a final solution $\mathbf{y}^{\text{fin}}$. This is the approximate solution for time-step k, i.e. $\mathbf{s}^k := \mathbf{y}^{\text{fin}}$ that we originally set out to find in Equation (5). It is then in turn used as the initial guess to compute an approximate solution for the next time-step $k + 1$.

Code Walkthrough

The provided code for the project already contains most of the functionalities described above, a brief overview of the code is presented in the this section. The main task of this project will be (i) to complete some parts of the sequential code and (ii) the parallelization of the code using OpenMP. This project will also serve as an example for using the message-passing interface MPI in project 4. There are three files of interest:

- `main.cpp`: initialization and main time-stepping loop
- `linalg.cpp`: the BLAS level 1 (vector-vector) kernels and conjugate gradient solver



(a) Interior grid points

(b) Boundary grid points

Figure 2: Visualization of stencil operators. The blue grid points indicate the grid points on the boundary of the domain with fixed values (see right side, set to zero in our setting). The orange grid points indicate inner grid points of the domain, whose stencil does not depend on the boundary values. The green grid points are still inside the domain but their stencils require boundary values.

  - This file defines simple kernels for operating on 1D vectors, including
    - dot product: $x \cdot y$: `hpc_dot()`
    - linear combination: $z = \text{alpha} * x + \text{beta} * y$: `hpc_lcomb()`
    - ...
  - All the kernels of interest in this HPC Lab for CSE start with `hpc_xxxxx`
- `operators.cpp`: the stencil operator for the finite difference discretization
  - This file has a function/subroutine that defines the stencil operator
  - The stencil operators differ depending on the position of the grid point, see Figures (2) and Algorithms 1 and 2:

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

Algorithm 1: Stencil: interior grid points

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

Algorithm 2: Stencil: boundary grid points

Compile and run the PDE mini-app on the ICS cluster

Log-in to the ICS cluster and afterwards load the gcc and python modules.

```
[user@login]$ module load gcc python
```

Go to the Project4 directory and use the makefile to compile the code

```
[user@login]$ make
```

Run the application on a compute node with selected parameters, e.g. domain size 128×128, 100 time steps and simulation time $0 - 0.01$ s, for now using a single OpenMP thread.

```
[user@login]$ salloc
[user@icsnodeXX]$ export OMP_NUM_THREADS=1
[user@icsnodeXX]$ ./main 128 100 0.01
```

After you implement the first part of the assignment, the output of the mini-application should look like this:

```
================================================================================
Welcome to mini-stencil!
version   :: C++ Serial
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.01
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 1.06824 seconds
7703 conjugate gradient iterations, at rate of 7210.92 iters/second
866 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

1. Task: Implementing the linear algebra functions and the stencil operators [40 Points]

The provided implementation is a serial version of the PDE mini-application with some code missing. Your first task is to implement the missing code to get a working PDE mini-application.

- Open the file `linalg.cpp` and implement the functions `hpc_xxxxx()`. Follow the comments in the code as they are there to help you with the implementation.
- Open file `operators.cpp` and implement the missing stencil kernels.

After completion of the above steps, the mini-app should produce correct results. Compare the number of conjugate gradient iterations and the Newton iterations with the reference output above. If the numbers are about the same, you have probably implemented everything correctly. Now try to plot the solution with the script `plotting.py`. It should look like in Figure 3.

```
$ ./plotting.py
```

Note

The script `plotting.py` reads the results computed by `./main` and creates an image `output.png`.
If X11 forwarding is enabled, it can also show the image in a window. To enable X11 forwarding, connect to ICS cluster with parameter `-Y`

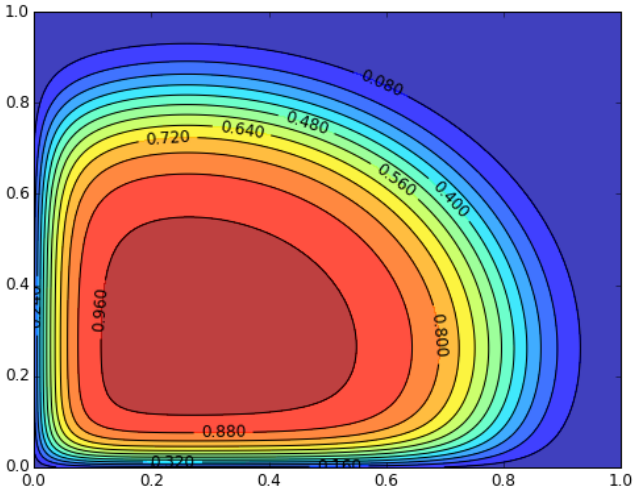


Figure 3: Output of the mini-app for a domain discretization into 128×128 grid points, 100 time steps with a simulation time from $0 - 0.01$ s

```
$ ssh -Y icsmaster
```

Then allocate a compute node with X11 forwarding and execute the plotting script on the compute node

```
$ salloc --x11
```

Note that to use the X11 forwarding, you need X server installed on your computer. On most Linux distributions it is already installed. On MacOS you have to download and install XQuartz and on Windows you have to download and install Xming. After the installation, you might need to reboot your computer.

If the plotting script prints an error

```
No module named 'matplotlib'
```

you probably forgot to load the `python` module. Please call

```
$ module load python
```

2. Task: Adding OpenMP to the nonlinear PDE mini-app [60 Points]

When the serial version of the mini-app is working we can add OpenMP directives. This allows you to use all cores on one compute node. In this project 3, you will measure and improve the scalability of your PDE simulation code. Scalability is the changing performance of a parallel program as it utilizes more computing resources. In this project we are interested in a performance analysis of both **strong scalability** and **weak scalability**. Strong scaling is identifying how a threaded PDE solver gets faster for a fixed PDE problem size. That is, we have the same discretization points per direction, but we run it with more and more threads and we hope/expect that it will compute its result faster. Weak scaling speaks to the latter point: how effectively can we increase the size of the problem we are dealing with? In a weak scaling study, each compute core has the same amount of work, which means that running on more threads increases the total size of the PDE simulation.

Replace welcome message in main.cpp

Replace the welcome message in `main.cpp` with a message that informs the user about the following points:

- That this is the OpenMP version.
- The number of threads it is using.

For example:

```
[user@login]$ salloc
[user@icsnodeXX]$ export OMP_NUM_THREADS=8
[user@icsnodeXX]$ ./main 128 100 0.01
================================================================================
Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 8
...
```

Linear algebra kernel

Open `linalg.cpp` and add OpenMP directives to all functions `hpc_XXXX()`, except for `hpc_cg()`.

- Recompile frequently and run with 8 threads to check that you are getting the right answer.

Once finished with this file, did your changes make any performance improvement?

- Compare the 128×128 and 256×256 results.

The diffusion stencil

The final step is to parallelize the stencil operator in `operators.cpp`,

- The nested for loop is an obvious target.
- It covers inner grid points.
- How about the boundary loops?

2.1. Strong scaling

Before starting adding OpenMP parallelism, find two sets of input parameters in term of larger grid size that converge for the serial version.

- You can use these parameters, or choose anything you like,
  - `./main 256 100 0.01`
  - `./main 512 100 0.01`
- Write down the time to solution.
  - You want this to get faster as you add OpenMP.
  - But you might have to add a few additional directives (such as SIMD instruction) before things actually get faster.
- Write down the number of conjugate gradient iterations.
  - Use this to check after you add each directive that you are still getting the right answer.
  - Remember that there will be some small variations because floating point operations are not commutative.
  - Review your OpenMP code and argue if a threaded OpenMP PDE solver can be implemented which produces bitwise-identical results without any parallel side effects.

How does it scale at different resolutions? Plot time to time to solution for the grid sizes below for 1-10 threads .

- 64×64
- 128×128
- 256×256
- 512×512
- 1024×1024

2.2. Weak scaling

Produce a weak scaling plot by running the PDE code with different numbers of threads and a fixed constant grid size per thread. When performing this weak-scaling study, we are asking a complementary question to that asking in a strong-scaling study. Instead of keeping the problem size fixed, we increase the problem size relative to the number of cores. For example, we might initially run a PDE simulation on a grid of 64×64 using 4 cores. Later, we might want to compare 128×128 using 16 cores. Will the time remain constant, since the work per core has remained the same, or will the time increase due, for example, to increased communications overhead associated with the larger problem size, or number of cores, or a higher number of nonlinear iterations? A weak scaling study is one method of documenting this behavior for your application, allowing you to make an guess at the amount of computing time you need.

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and observations for all exercises by writing an extended Latex report. Use the Latex template provided on the webpage and upload the Latex summary as a PDF to iCorsi .

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your OpenMP solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .zip/.tgz through iCorsi .