Università della Svizzera italiana
Institute of Computing CI

High-Performance Computing, Fall 2025
Lecturer: Dr. A. Eftekhari
Assistants: V. Karpenko, L. Migliari, D. Vega

# Project 3
# Parallel Space Solution of a Nonlinear PDE using OpenMP

**Due date**: See iCorsi submission

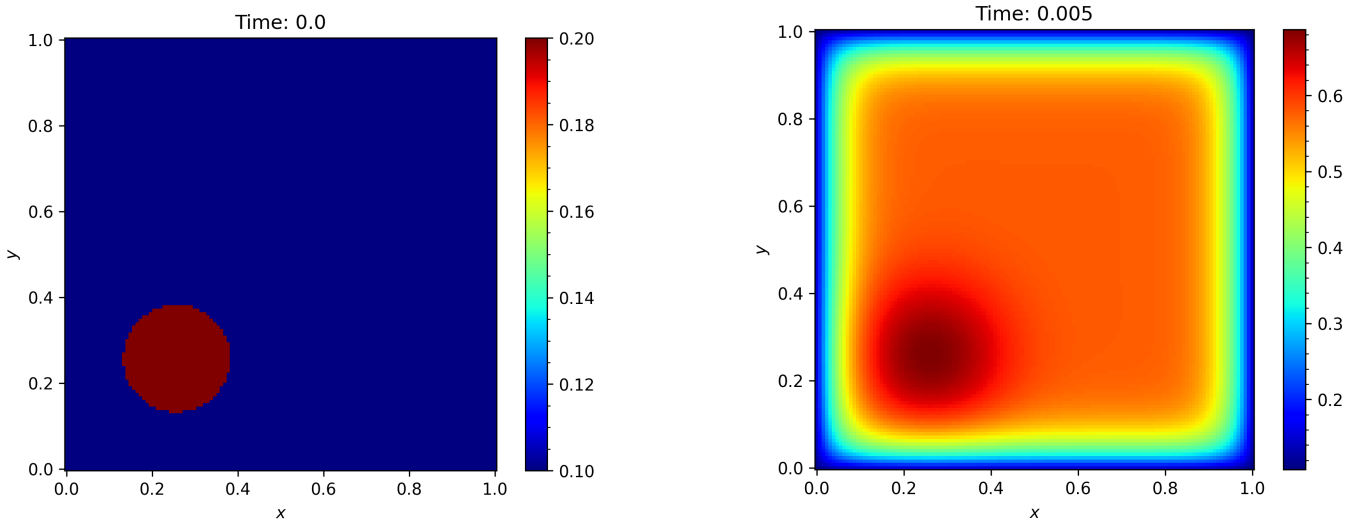
In this project, we will develop a parallel PDE mini-app using `OpenMP`. `OpenMP` seems to be a straightforward way to parallelize (incrementally) a given application by simply adding directives to compute intensive loops. However, it turns out that getting a truly scalable `OpenMP` application is far from trivial in many cases, especially on today's CPUs featuring dozens of cores. The idea of this project is to confront you (in a friendly manner, of course) with this unfortunate truth. You may do this project in groups of students (max four or five). In fact, we prefer that you do so.
For the whole project, the simulations must be run on the Rosa cluster. However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers, but please make sure to document them in your report if you include results.
You find all the skeleton source codes for the project on the course iCorsi page.
To compile your code, we recommend using a `makefile` (either provided or from other in-class examples) and adjusting them as needed. While developing/debugging your code, it can be useful to work in an interactive session (you can use the `-reservation=hpc-monday` or `-reservation=hpc-thursday` for better priority) as follows:

```
[user@icslogin01 Test]$ srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i
srun: job 9737 queued and waiting for resources
srun: job 9737 has been allocated resources
[user@icsnodeXX Test]$ export OMP_NUM_THREADS=2
[user@icsnodeXX Test]$ ./hello_omp
OpenMP threads=2
```

This allocates an interactive session on 1 node of the Rosa cluster for 1 minutes. The program sets the number of threads for parallel regions according to the value of the environment variable `OMP_NUM_THREADS` which is assigned before executing the code.
**Note**: Using `--exclusive` grants you exclusive access to the node, meaning that there is no sharing of resources—its all yours! It is very important to use exclusive allocation sparingly, as it restricts others from accessing the node. For code compilation and testing, it is recommended to use `srun --nodes=1 --cpus-per-task=4 --time=00:20:00 --pty bash -i` (where `--cpus-per-task` sets the maximum number of threads). When you are ready to run larger tests and collect results, such as strong scaling results, you would use `srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i`.

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

The simple `OpenMP` exercises such as the $\pi$-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 this project, we are going to implement a parallel PDE mini-app that solves something more sophisticated, but where the code is nevertheless still relatively short and easy to understand. Our mini-app will solve a prototypical reaction-diffusion equation with the finite difference method. Although the simplicity, it is an example of so-called stencil-based kernels that constitute the core of many important scientific applications on block-structured grids, including numerical climate modeling, cosmological simulations and many more in between.




(a) (b)

Figure 1: The population concentration at time $t = 0$ (left) and $t = 0.005$ (right).

We consider Fisher's equation that can be used to simulate simple population dynamics. In two-dimensional Cartesian coordinates it is given by

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

where $s = s(x, y, t)$ is the population concentration, $D$ is the diffusion constant and $R$ is the reaction constant. The left hand side represents the rate of change of $s$ over time. On the right hand side, the first term describes the diffusion of $s$ in space and the second term describes the growth of the population.
We consider Eq. (1) on a square Cartesian domain $(x, y) \in \Omega = [0, 1]^2$ with Dirichlet boundary conditions

$$s(x, y, t) = 0.1 \quad \text{for } (x, y) \in \partial\Omega \tag{2}$$

and the initial conditions

$$s(x, y, t = 0) = \begin{cases} 0.2 & \text{for } (x - x_c)^2 + (y - y_c)^2 < r^2, \\ 0.1 & \text{elsewhere,} \end{cases} \tag{3}$$

where $x_c = y_c = \frac{1}{4}$ and $r = \frac{1}{8}$. The initial conditions and the population concentration at time $t_f = 0.005$ are displayed in Fig. 1.
To discretize the domain $\Omega$, we introduce a uniform grid composed of $(n+2) \times (n+2)$ grid points. The grid points are denoted as $(x_i, y_j)$, where $x_i = i\ h$ and $y_j = j\ h$ for $i, j = 0, 1, \ldots, n+1$ and $h = \frac{1}{n+1}$ is the uniform grid spacing. Here, $x_0 = 0$ and $x_{n+1} = 1$ define the boundaries along the $x$-axis, and similarly, $y_0 = 0$ and $y_{n+1} = 1$ define the boundaries along the $y$-axis. The discretization is shown in Fig. 2. Likewise, we discretize time into steps $k = 0, \ldots, n_t$ of size $\Delta t = \frac{t_f}{n_t}$. We denote by $s_{i,j}^k$ the approximation of the population concentration at the grid point $(x_i, y_j)$ at time step $k$ (i.e., $s_{i,j}^k \approx s(x_i, y_j, t^k)$).
We use a second-order finite difference discretization to approximate the spatial derivatives of $s$ for all inner grid points, i.e.,

$$\begin{aligned} \left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right)_{i,j} &\approx \frac{s_{i+1,j} - 2s_{i,j} + s_{i-1,j}}{h^2} + \frac{s_{i,j+1} - 2s_{i,j} + s_{i,j-1}}{h^2} \\ &= \frac{1}{h^2}(-4s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1}), \end{aligned} \tag{4}$$

for all grid point $(i, j) \in \{1, \ldots, n\}$. This results in the so-called 5-point stencil and it is displayed in Fig. 2. Note that one distinguishes interior grid points, where we seek an approximate solution, and boundary grid points, which are fixed by the Dirichlet condition. In order to approximate the time derivative, we use a first-order implicit Euler difference scheme, which at

time step $k$ gives

$$\left(\frac{\partial s}{\partial t}\right)^k_{i,j} \approx \frac{1}{\Delta t}(s^k_{i,j} - s^{k-1}_{i,j}). \tag{5}$$

Putting together the components from Eqs. (4) and (5), we obtain the following discretization of Eq. (1):

$$\frac{1}{\Delta t}(s^k_{i,j} - s^{k-1}_{i,j}) = \frac{D}{h^2}\left(-4s^k_{i,j} + s^k_{i-1,j} + s^k_{i+1,j} + s^k_{i,j-1} + s^k_{i,j+1}\right) + Rs^k_{i,j}(1 - s^k_{i,j}), \tag{6}$$

which we will attempt to solve in order to obtain an approximate solution for the population concentration $s$. By moving all terms on the right-hand side and multiplying with $h^2/D$, we can reformulate Eq. (6) as

$$f^k_{i,j} = \left[-(4+\alpha)s^k_{i,j} + s^k_{i-1,j} + s^k_{i+1,j} + s^k_{i,j-1} + s^k_{i,j+1} + \beta s^k_{i,j}(1 - s^k_{i,j})\right] + \alpha s^{k-1}_{i,j} = 0 \tag{7}$$

for each grid point $(i, j)$ and time step $k$ with $\alpha = h^2/(D\Delta t)$ and $\beta = Rh^2/D$.
At time step $k = 1$, we initialize $s^{k-1}_{i,j} = s^0_{i,j} = s(x, y, t = 0)$, and we look for approximate values $s^k_{i,j}$ that fulfill Eq. (7). For all $(i, j)$ at a fixed $k$, 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 Eq. (7). 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 vector containing 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 at the $l$-th iteration

$$\mathbf{y}^{(l+1)} = \mathbf{y}^{(l)} - [\mathbf{J_f}(\mathbf{y}^{(l)})]^{-1}\mathbf{f}(\mathbf{y}^{(l)}), \tag{8}$$

where $\mathbf{J_f}(\mathbf{y}^{(l)}) \in \mathbb{R}^{N\times N}$ 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 $[\mathbf{J_f}(\mathbf{y}^{(l)})]^{-1}$ is not readily available. We do not compute it directly but instead use a matrix-free Conjugate Gradient (CG) solver that solves the following linear system of equations for $\delta\mathbf{y}^{(l+1)} = \mathbf{y}^{(l+1)} - \mathbf{y}^{(l)}$:

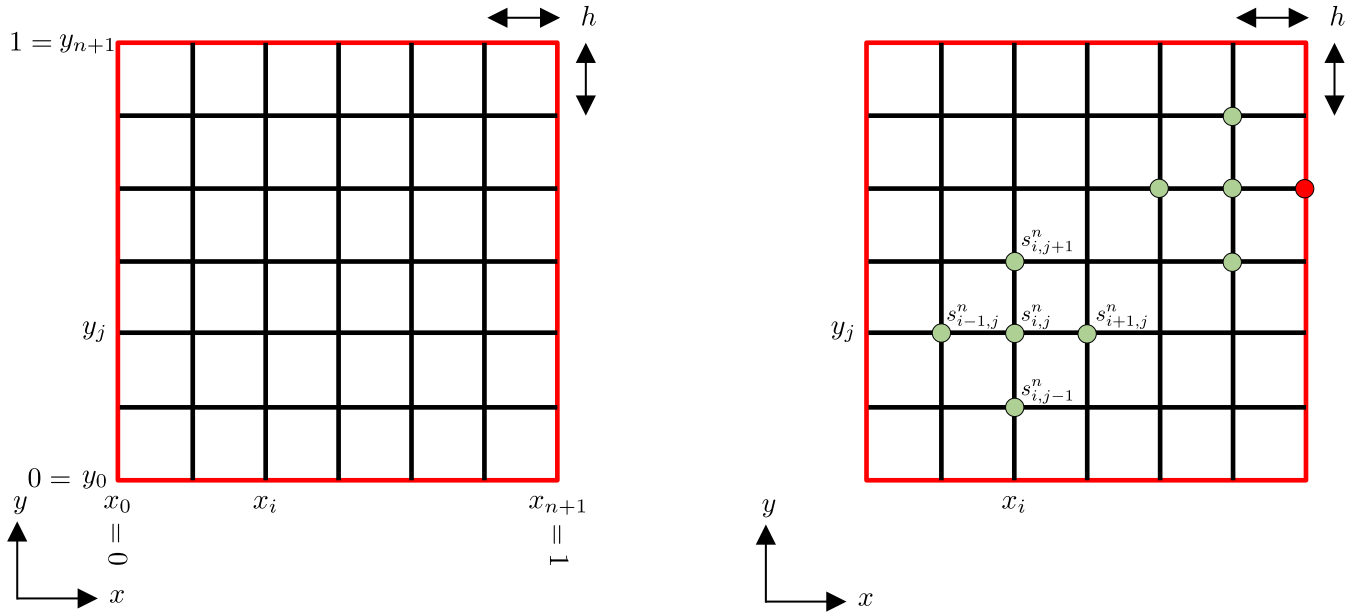
$$[\mathbf{J_f}(\mathbf{y}^{(l)})]\delta\mathbf{y}^{(l+1)} = -\mathbf{f}(\mathbf{y}^{(l)}) \tag{9}$$

We iterate over $l$ in Eq. (8) 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 Eq. (7). 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 on the iCorsi page for the project already contains most of the functionalities described above, a brief overview of the code is presented in 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 another project. There are three files of main interest:

- `main.cpp`: initialization and main time stepping loop.
- `linalg.cpp`: the BLAS level 1 (vector-vector) kernels and conjugate gradient solver. All the kernels of interest in this HPC Lab for CSE start with `hpc_xxxxx`.
  - Scalar product $x \cdot y$: `hpc_dot()`.
  - Linear combination $z = \text{alpha} * x + \text{beta} * y$: `hpc_lcomb()`.
  - ...
- `operators.cpp`: the stencil operator for the finite difference discretization.




(a) (b)

Figure 2: The left panel shows the discretization of the domain $\Omega = [0, 1]^2$. The right panel shows a visualization of the stencil operators.

## Compile and run the PDE mini-app on the Rosa cluster

Use the `Makefile` to compile the mini-app:

```
[user@icslogin01]$ module load gcc
[user@icslogin01]$ make
```

Run the application on a compute node with selected parameters, e.g. domain size $128 \times 128$, 100 time steps and simulation time $t = 0 - 0.005$ s:

```
[user@icslogin01]$ srun --nodes=1 --exclusive --time=00:01:00 --pty bash -i
srun: job 10589 queued and waiting for resources
srun: job 10589 has been allocated resources
[user@icsnodeXX]$ ./main 128 100 0.005
```

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

```
[user@icsnodeXX]$ ./main 128 100 0.005
================================================================================
Welcome to mini-stencil!
version   :: C++ Serial
threads   :: 1
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
```

```
simulation took 0.25046 seconds
1513 conjugate gradient iterations, at rate of 6040.89 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

The mini-app generates `output.bov` and `output.bin` files, which contain the population concentration at the final time. To visualize the data, follow these steps:

1. **Download the files**: You can either:
   - Use `scp` to transfer `output.bov` and `output.bin` from the cluster to your local machine. Here's an example command:

     ```
     scp user@rosa.usi.ch:/path/to/output.bov
      ↪  user@rosa.usi.ch:/path/to/output.bin /local/destination/directory/
     ```

     Replace `user` and `/path/to/` with your actual cluster username and file path. Also, specify your desired local directory.
   - Use `sftp` in the same fashion as the projects before this one.
2. **Install necessary Python packages**: On your local machine, ensure you have Python, NumPy, and Matplotlib installed. If not, you can install them with:

   ```
   pip install numpy matplotlib
   ```

3. **Generate the plot**: Run the plotting script on your local machine using:

   ```
   python plot.py
   ```

   Make sure `plot.py`, `output.bov`, and `output.bin` are in the same directory, or adjust the paths accordingly.

For additional information on using Python, NumPy, and Matplotlib, please refer to their respective documentation.
This creates `output.png` showing the population concentration at final time. It should look like Fig. 1b.

# 1 Task: Implementing the linear algebra functions and the stencil operators [35 Points]

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

- Implement the functions `hpc_xxxxx()` in `linalg.cpp`. Follow the comments in the code as they are there to help you with the implementation. [18 Points]
- Implement the missing stencil kernel in `operators.cpp`. [15 Points]

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.

- Plot the solution with the script `plot.py` and include it in your report. It should look like in Figure 1b. [2 Points]

# 2 Task: Adding OpenMP to the nonlinear PDE mini-app [50 Points]

When the serial version of the mini-app is working we add OpenMP directives. In this project, 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. 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.

## 2.1 Welcome message in `main.cpp` and serial compilation [2 Points]

Replace the welcome message in `main.cpp` with a message that informs the user about (i) that the code is using OpenMP and (ii) the number of threads:

```
[user@icsnodeXX]$ export OMP_NUM_THREADS=4
[user@icsnodeXX]$ ./main 128 100 0.005
================================================================================
Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 4
...
```

Make sure that the code still compiles without OpenMP enabled.
**Hint**: Assume that the supported compilers feature the `_OPENMP macro` specifications[1].

## 2.2 Linear algebra kernel [15 Points]

Add OpenMP directives to parallelize all loops in the functions `hpc_XXXX()`, except for `hpc_cg()`.

## 2.3 The diffusion stencil [10 Points]

Add OpenMP directives to parallelize the stencil operator in `operators.cpp`. The nested for loop and the inner grid points might be obvious targets. What role do the boundary loops play?

## 2.4 Bitwise identical results [3 Points]

Argue if your threaded OpenMP PDE solver can be implemented so that it produces bitwise-identical results (i.e., without any parallel side effects or not).
**Hint**: Recall that floating point addition/multiplication operations are not associative and the note on parallel reduction in the OpenMP specifications[2].

## 2.5 Strong scaling [10 Points]

How does your code scale for different resolutions? Plot the time to solution for $N_{\text{CPU}} = 1, 2, 4, 8, 16$ threads across resolutions of $n \times n$, where $n = 64, 128, 256, 512, 1024$. Interpret your results.
For example: For a resolution $n = 64$, plot the time to solution for $64 \times 64$ on $N_{\text{CPU}} = 1, 2, 4, 8, 16$ threads.

[1] https://www.openmp.org/spec-html/5.2/openmpse16.html#x47-460003.3
[2] https://www.openmp.org/spec-html/5.2/openmpsu50.html

### 2.6 Weak scaling [10 Points]

How does code scale for constant work by threads ratio? Plot the time to solution as the total problem size and the number of threads $N_{\mathrm{CPU}} = 1, 2, 4, 8, 16$ increase proportionally, to maintain a constant workload per thread, and the base resolutions $n \times n$ and $n = 64, 128, 256$. Interpret your results.
For incremental scaling, you can calculate the problem size for each thread count $N_{\mathrm{CPU}}$ using $n =$ base resolution $\times \sqrt{N_{\mathrm{CPU}}}$ to maintain a consistent workload per thread.
For example: For base resolution $n = 64$, plot the time to solution for $64 \times 64$ on $N_{\mathrm{CPU}} = 1$, $91 \times 91$ on $N_{\mathrm{CPU}} = 2$, $128 \times 128$ on $N_{\mathrm{CPU}} = 4$, $181 \times 181$ on $N_{\mathrm{CPU}} = 8$, and $256 \times 256$ on $N_{\mathrm{CPU}} = 16$.
**Hint**: Keep in mind that the convergence of the nonlinear solver might depend on the resolution.

## 3 Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

## 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 the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a compressed archive, formatted like project_number_lastname_firstname.zip/.tgz . It should contain:
  - All the source codes of your solutions.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up with your name.
  - Follow the provided guidelines for the report.
- Submit your .tgz/.zip through iCorsi.

## Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.