High-**P**erformance **C**omputing 20**24**

Basics of Numerical Methods for PDEs

Aryan Eftekhari, Olaf Schenk | **U**niversità della **S**vizzera **i**taliana (USI) | **I**nstitute of **C**omputing (CI)

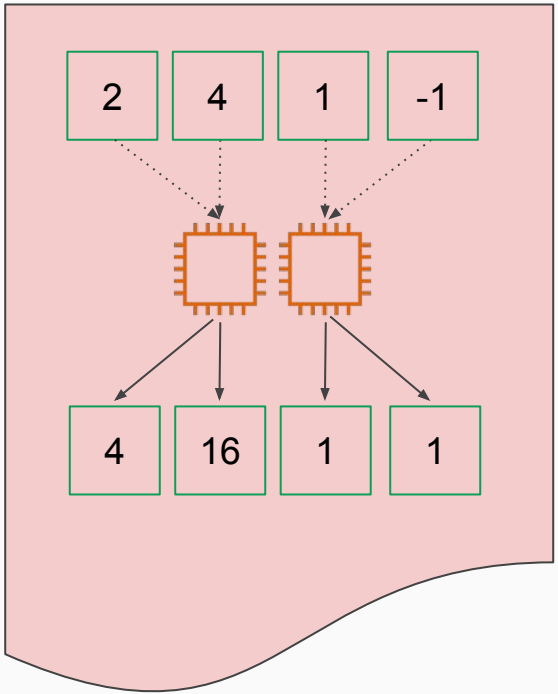
SIMD and **Memory Hierarchy** are fundamental to modern computing systems.

Review
OpenMP

Shared Memory Parallelization

*Computation is distributed along **threads**.*

Synchronization between threads.



OpenMP is easy to use …

```
#include <omp.h>
#include <vector>

int main(){
    std::vector<double> val(1e8,0);
    #pragma omp parallel for
    for (int i = 0; i < val.size(); i++)
        val[i] = COSTLY_OPERATION(i);
    return 0;
}
```

Parallel Region

```
// In Terminal/Command line
// Compile via command line (or makefile)
g++ -fopenmp -O3 main.cpp -o main.exe
// Run
export OMP_NUM_THREADS=2; ./main.exe
```

Overview

A **brief** and **basic** overview

of the numerical methods for solving PDEs.

Differential Equations
Ordinary and Partial

Ordinary Differential Equation (ODE)

Differentiation is with respect to *one variable*.

For example, exponential growth and decay, and Newton's second law of motion.

$$\frac{d^2y}{dx^2} - 3\frac{dy}{dx} + 2y = x^2$$

Partial Differential Equation (PDE)

Differentiation is with respect to *more than one variable*.

For example, heat equation, wave equation, and Fisher's equation.

$$\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2} = \alpha\frac{\partial s}{\partial t}$$

Multivariate Calculus
Notation and Jargon

Gradient
Differentiation of scalar valued function with respect to a vector.

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n}\right]^\top$$

Jacobian
Differentiation of vector valued function with respect to more than one variables (i.e., vector).

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Hessian
Second-order differentiation of scalar valued function with respect to more than one variable .

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Laplacian (operator)
Divergence of the gradient or vector field (i.e., trace of the Hessian).

$$\Delta f = \sum_i^n \frac{\partial^2 f}{\partial x_i^2}$$

Solution Method
Basic Steps

1. **Formulation and representation**

Define the mathematical model of the problem including the domain, **initial condition**, **boundaries conditions**, and governing equations.*

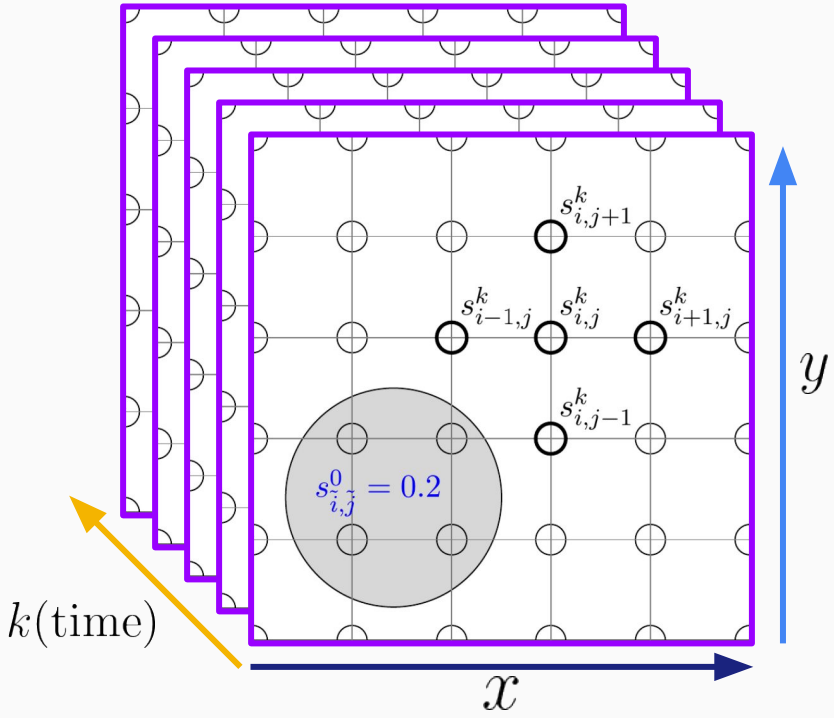
2. **Discretization in space and time**

Convert the continuous problem into a set of discrete equations using a chosen numerical method.

3. **Solve discretized problem in space and time**

Compute the solution of the discrete system over the defined domain.

**consider also stability of the solution.*



Representation of space and time for 2D.

Solution Method

Formulation/Discretization, e.g., Fisher's Equation

$$\frac{\partial s}{\partial t} = \delta \Delta s + \rho s(1 - s)$$

Used to describe biological populations: **spatial diffusion** with **reaction/growth**.

$$\frac{1}{\tau}(s_{i,j}^{k} - s_{i,j}^{k-1}) = \underbrace{\frac{\delta}{h^2}\left(-4s_{i,j}^{k} + s_{i+1,j}^{k} + s_{i-1,j}^{k} + s_{i,j+1}^{k} + s_{i,j-1}^{k}\right)}_{\text{linear}} + \underbrace{\rho s_{i,j}^{k}(1 - s_{i,j}^{k})}_{\text{nonlinear}}$$

Boundary Conditions: What if the (i,j) is at the edge of the grid ?

Initial Condition: We always need the previous solution!

Solution Method

Solve, e.g., Fisher's Equation

$$\frac{1}{\tau}(s_{i,j}^k - s_{i,j}^{k-1}) = \underbrace{\frac{\delta}{h^2}\left(-4s_{i,j}^k + s_{i+1,j}^k + s_{i-1,j}^k + s_{i,j+1}^k + s_{i,j-1}^k\right)}_{\text{linear}} + \underbrace{\rho s_{i,j}^k(1 - s_{i,j}^k)}_{\text{nonlinear}}$$

Let $\mathbf{s}^k := [s_{i,j}^k, s_{i,j+1}^k, s_{i,j+2}^k, \ldots, s_{i+1,j}^k, s_{i+2,j}^k, \ldots]^\top$

The solution is than the **root** of :

$$f(\mathbf{s}^k | \mathbf{s}^{k-1}, \mathbf{A}, c_1, c_2) := \mathbf{s}^k - \mathbf{s}^{k-1} - c_1\mathbf{A}\mathbf{s}^k - c_2\mathbf{s}^k \cdot (1 - \mathbf{s}^k)$$

Solution Method

Solve, e.g., Fisher's Equation

Newton Iteration–A Method for root finding:

$$\mathbf{s}^k \leftarrow \mathbf{s}^k - [\mathbf{J}_f]^{-1} f(\mathbf{s}^k)$$

Remark: We omit the "given" variables in the notation for clarity.

We need to solve a linear system of equations!

$$[\mathbf{J}_f]^{-1} f(\mathbf{s}^k) = \mathbf{x} \iff f(\mathbf{s}^k) = [\mathbf{J}_f]\mathbf{x}$$

Solution Method
Pseudo Algorithm

```
Input s_initail_value, K, iter_max, eps

// Initial Conditions
s_last ← s_initail_value
s      ← s_last

// Time loop
For k = 1 to K
      // Newton loop
      For iter=1 to iter_max
            // Linear Solve (will have its own loop)
            update ← lin_solve(J(s|s_last),f(s|s_last))
            s ← s - update
            // Convergence Check
            If norm(update)<eps
            break
            Endif
      Endfor
      // Swap Solution
      s_last ← s
Endfor

Return s
```

Common Options for `lin_solve`:

1. Direct Methods

Solve matrices in fixed steps with notable stability, especially for well-conditioned systems.

2. Iterative Methods

Memory-efficient with *adjustable accuracy,* though they demand careful considerations for stability.

Note: Iterative methods can be implemented in a matrix-free manner and rely on easily parallelizable operations.

As before, the solution is than the **root** of:

$$f(\mathbf{s}^k|\mathbf{s}^{k-1}, \mathbf{A}, c_1, c_2) := \mathbf{s}^k - \mathbf{s}^{k-1} - \underbrace{c_1\mathbf{A}\mathbf{s}^k - c_2\mathbf{s}^k \cdot (1 - \mathbf{s}^k)}$$

What if we use $\mathbf{s}^{k-1}$ in place of $\mathbf{s}^k$?
… we get an "*explicit method*".

Explicit vs Implicit Methods

- Explicit Methods (in the above)
  No need to solve a system of equations, making them much easier to program.
  The solution can be unstable (may diverge), making them unsuitable for many serious applications.

- Implicit methods (what we showed before)
  Require solving a system of equations, increasing programming complexity.
  The solution is stable, making implicit methods essential for many challenging problems.

Università della Svizzera italiana
Institute of Computing CI

High-Performance Computing, Fall 2024
Lecturer: Dr. A. Eftekhari, Prof. O. Schenk
Assistants: M. Lechekhab, D. Vega,
D. Santarsiero, J. Palumbo, I. Ecevi

Project 3
Parallel Space Solution of a Nonlinear PDE using OpenMP

Due date: 30 October 2024 at 23:59 (See iCorsi for updates)

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-tuesday` or `-reservation=hpc-wednesday` 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.

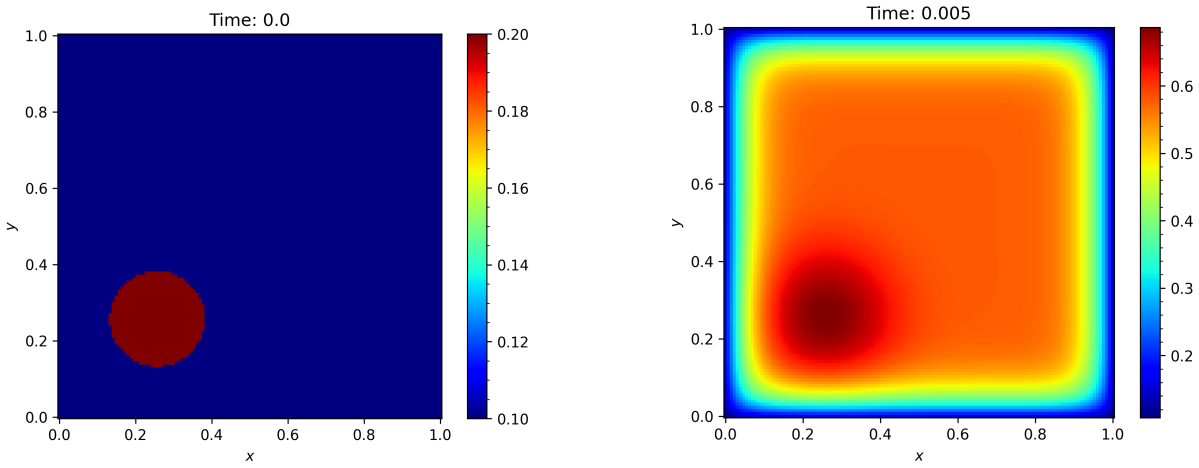
Note: Using `--exclusive` grants you exclusive access to the node, meaning that there is no sharing of resources—its all yours! It is very important to use exclusive allocation sparingly, as it restricts others from accessing the node. For code compilation and testing, it is recommended to use `srun --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 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 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.

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

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



(a) (b)

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

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 Ω, 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 $s^k_{i,j}$ the approximation of the population concentration at the grid point (x_i, y_j) at time step k (i.e., $s^k_{i,j} \approx s(x_i, y_j, t^k)$).

We use a second-order finite difference discretization to approximate the spatial derivatives of s for all inner grid points, i.e.,

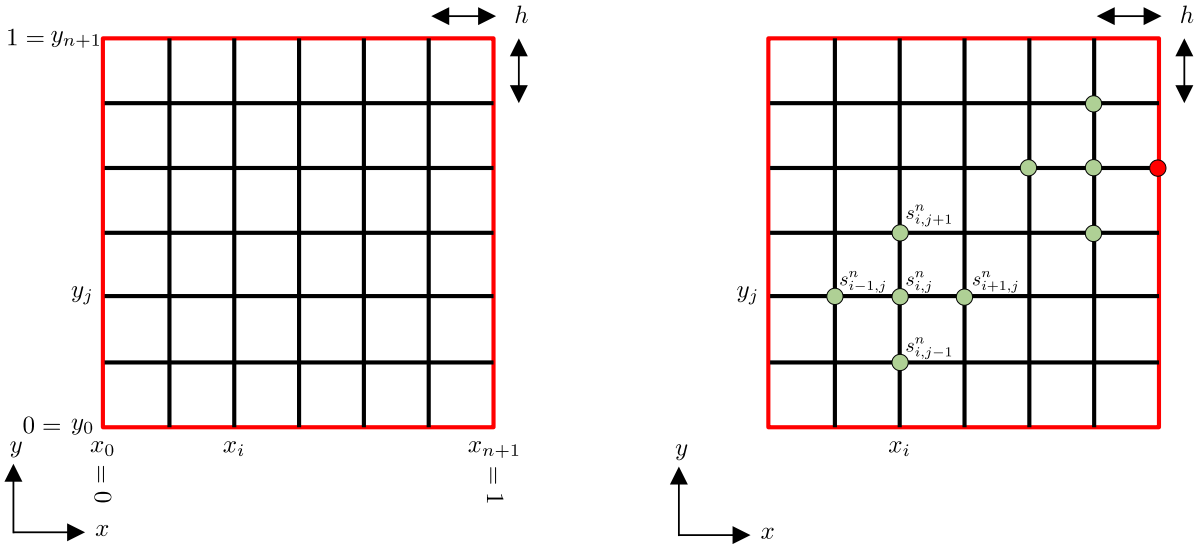
$$\begin{aligned} \left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right)_{i,j} &\approx \frac{s_{i+1,j} - 2s_{i,j} + s_{i-1,j}}{h^2} + \frac{s_{i,j+1} - 2s_{i,j} + s_{i,j-1}}{h^2} \\ &= \frac{1}{h^2}(-4s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1}), \end{aligned}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

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):

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



(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.

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.

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 128×128, 100 time steps and simulation time $t = 0 - 0.005\,\text{s}$:

```
[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.
   - Alternatively, if you are using an IDE with remote file access (e.g., VS Code), you can directly copy the files from the cluster to your local machine using the IDE's file explorer.

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 64×64 on $N_{\text{CPU}} = 1, 2, 4, 8, 16$ threads.

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 $N_{\text{CPU}} = 1, 2, 4, 8, 16$ 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_{CPU} using $n =$ base resolution $\times \sqrt{N_{\text{CPU}}}$ to maintain a consistent workload per thread.
For example: For base resolution $n = 64$, plot the time to solution for 64×64 on $N_{\text{CPU}} = 1$, 91×91 on $N_{\text{CPU}} = 2$, 128×128 on $N_{\text{CPU}} = 4$, 181×181 on $N_{\text{CPU}} = 8$, and 256×256 on $N_{\text{CPU}} = 16$.
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 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 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 through iCorsi.

[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

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.

Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 3

This project will introduce you a parallel space solution of a nonlinear PDE using OpenMP.

1. Implementing the linear algebra functions and the stencil operators [35 Points]

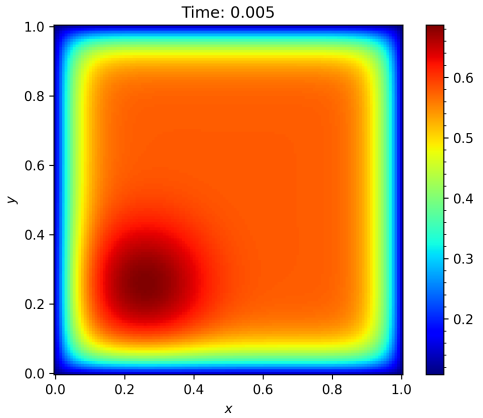
After implementing the missing functions and the stencil kernel, given by the formula 1, the serial code produces the output seen in Subfigure 1a, which is approximately the same as the provided reference output in the project description.

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

Furthermore the resulting population concentration at final time is depicted in Subfigure 1b.

```
[huberde@icsnode30 mini_app]$ ./main.exe 128 100 0.005
================================================================================
                        Welcome to mini-stencil!
version   :: C++ Serial
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.224838 seconds
1511 conjugate gradient iterations, at rate of 6720.39 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 1, 128, 100, 1511, 300, 0.224838 ###
Goodbye!
```

(a) Output after implementing the linear algebra function and stencil operators.



(b) Population concentration at $t = 0.005$.

Figure 1: Results of the Mini App, when implemented in serial.

2. Adding OpenMP to the nonlinear PDE mini-app [50 Points]

2.1. Implementation

In order to ensure that the mini-app still compiles without OpenMP enabled we introduce the following macro

```
#ifdef _OPENMP
/* CODE */
#else
/* CODE */
#endif
```

Listing 1: _OpenMP macro

which functions similar to an `if` statement, that is evaluated at compile time. All the necessary functions from the OpenMP that would conflict with the compilation process (which does not include the `#pragma omp` statements) are placed inside the `#ifdef` macro. [2] The updated welcome massage can be seen in Figure 2

```
[huberde@icsnode30 mini_app]$ export OMP_NUM_THREADS=8
[huberde@icsnode30 mini_app]$ ./main.exe 128 100 0.005
================================================================================
                      Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 8
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.101617 seconds
1516 conjugate gradient iterations, at rate of 14918.8 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 1, 128, 100, 1516, 300, 0.101617 ###
Goodbye!
```

Figure 2: Welcome Massage for parallel Implementation

For the parallelization of all the loops in the `linalg.c` file, I introduce basic OpenMP pragma for loops statements including the reduction clause, when necessary. No extra clauses were introduced, because these are rather simple loops, which OpenMP should be able to optimize very well on its own.

In contrast for the diffusion stencil different ways of optimizing the loop were experimented on. This included using the `collapse` clause for the inner grid points [1], trying out different loop scheduling (static, guided, dynamic), as well as parallelizing the loops for the boundary points. None of these approaches brought a significant improvement to the execution, therefore the decision was taken to resort to the simplest solution, which is simply to use `#pragma omp parallel for` statement for the inner grid points and let OpenMP do all the optimization.

2.2. Boundary Loops

The boundary loops play a essential role in diffusion simulations by appropriately handling boundary conditions. By enforcing boundary conditions they are maintaining the stability and correctness of the model. In the mini-app they are split up in East, West, North, South and Corners. Each boundary loop has a unique constraint that needs to taken into consideration, this includes handling points outside of the domain or including fixed boundary values.

2.3. Bitwise identical results

Producing a bitwise-identical result using more than one thread in OpenMP can be a challenge, because of the non-associativity of floating point addition or multiplications, which can lead to different results even when using the same number of threads. More specifically when using the reduction clause, OpenMP does not specify the location nor the order in which the values are combined. This can lead to different rounding errors, due to the limited accuracy of floating

point operations, depending on the order of the reduction [3]. Technically it would be possible to have bitwise-identical solution for a fixed number of threads, where we guarantee that always the same chunk of data is processed by a thread and then have a fixed order in which we reduce individual results, but as soon as we change the number of threads we can not guarantee a bitwise-identical solution. In addition this method would have a clear impact on performance, because these additional assumption we would need to take would create an additional overhead.

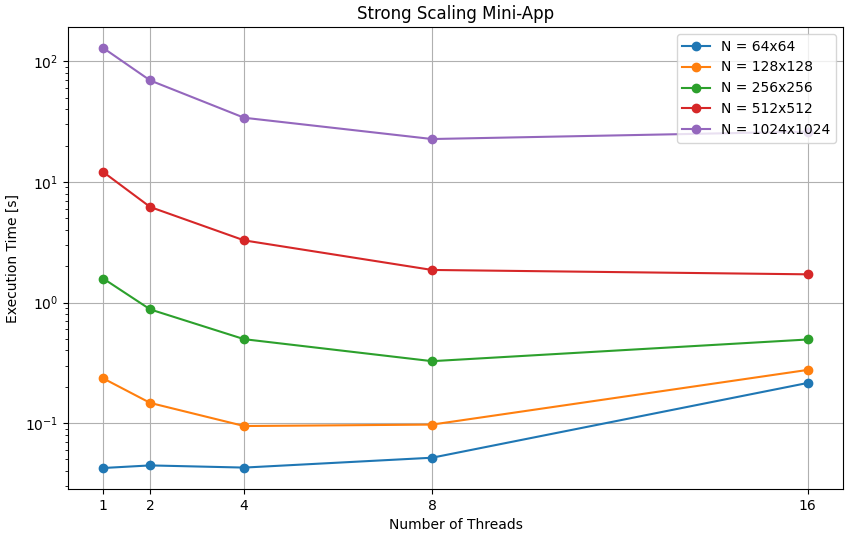
2.4. Strong & Weak Scaling Analysis

In the strong analysis scenario, where the problem size is constant and the goal is to reduce the execution time by increasing the number of threads. The Figure 3a depicts the execution time in seconds versus the number of threads, for the y-axis a logarithmic scale was chosen in order to examine the variations in timing across different resolutions.
For the smallest problem size in $N = 64 \times 64$ the curve is basically flat for 2 to 4 number of threads, increasing it even further the execution time starts to increase, which is caused by the increasing overhead created by OpenMP, when handling synchronization, communication and thread management. This clearly outweighs the benefits given by the parallelization. This behavior clearly indicates that for efficient use of the threads the work load is too small. Increasing the resolution, we can now clearly see a reduction of execution from threads 2 to 4 for the $N = 128 \times 128$ and even from threads 2 to 8 for the resolutions from $N = 256 \times 256$ to $N = 1024 \times 1024$. This clearly shows that all these resolution benefit from a increasing number of threads, yet when increasing to 16 threads for all the resolution the execution time increases or stays relatively stable (for higher resolutions). This is again caused by the increasing overhead. In addition this clearly shows that for all resolution a saturation point is reached, after which the benefits diminish.

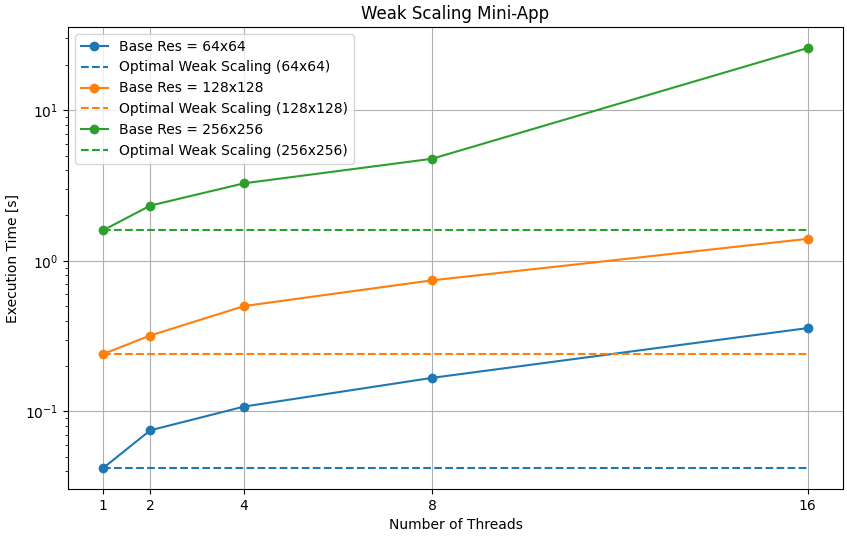
In the weak scaling scenario, when increasing the number of threads, the workload per thread stays constant. In the optimal scenario the execution time stays constant, when scaling up the number of threads and resolution. This resolution is adjusted by the following formula

$$n = n_{\text{base}} \cdot \sqrt{N_{\text{CPU}}} \tag{2}$$

If there is an increase in execution time this represents the overhead created by the communication, thread management and synchronization. The results of the mini-app using weak scaling can be seen in Figure 3b, where a logarithmic scale was used for the y-axis to compare the execution times more conveniently. In addition the dashed lines indicate the constant optimal execution time. Smaller base resolutions (64×64 and 128×128) clearly show less efficient weak scaling. For smaller number of threads the execution time increases significantly. Once more this shows the overhead encountered is substantial in comparison to the workload. This suggests that problem size might be too small to leverage parallel resources more effectively. For higher base resolution (256×256) weak scaling is improved, which can be seen by the more gradual grow when increasing the number of threads, suggesting that due to the higher resolutions the overhead is less dominant. When comparing curves for each base resolution to its optimal weak scaling, we can clearly make out a strong deviation, which becomes more noticeable when increasing the number of threads and as a consequence, also the overhead.



(a) Strong Scaling



(b) Weak Scaling

Figure 3: Scaling Plots of the mini-app

References

[1] *Collapse Clause*. URL: `https://www.openmp.org/spec-html/5.2/openmpsu30.html` (visited on 10/31/2024).

[2] *Conditional Compilation*. URL: `https://www.openmp.org/spec-html/5.2/openmpse16.html#x47-460003.3` (visited on 10/31/2024).

[3] *Reduction Scoping Clauses*. URL: `https://www.openmp.org/spec-html/5.2/openmpsu50.html` (visited on 10/31/2024).