ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE

FS 2024
Dr. R. Käppeli,
Dr. A. Schell, M. R. Herde,
C. Humbel, A. Mazzoleni,
N. A. Siegenheim

Project 3

Due date: Monday 15 April 2024, 11:59pm (midnight)

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 Euler (phase 2) nodes with AMD EPYC 7763 CPUs (for which we have privileged access during the lab hours), and we will use the new software stack (LMOD Modules) using the GNU Compiler Collection:

```
[user@eu-login-39 ~]$ module load gcc
[user@eu-login-39 ~]$ gcc --version
gcc (GCC) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

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 Moodle page.

While developing/debugging your code, it can be useful to work in an interactive session as follows:

```
[user@eu-login-39 ]$ srun --ntasks=1 --cpus-per-task=4 --mem-per-cpu=1024 \
                         --constraint=EPYC_7763                           \
                         --time=01:00:00 --pty bash
srun: Job step's --cpus-per-task value exceeds that of job (4 > 1). Job step may
↪  never run.
srun: job 50078034 queued and waiting for resources
srun: job 50078034 has been allocated resources
[user@eu-a2p-409 ]$
```

This allocates an interactive session on an Euler VII (phase 2) node with up to 4 cores and 4GB of memory for one hour[1].

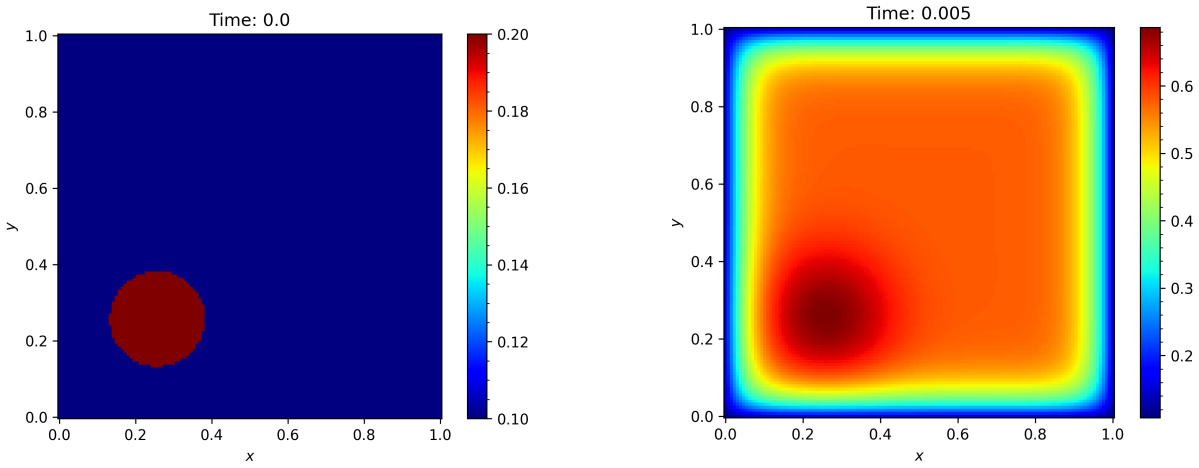
The Fisher's equation as an example of a reaction-diffusion PDE

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-appl 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

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

[1] We ignore the warning that the job might never run, it should.



(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):

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

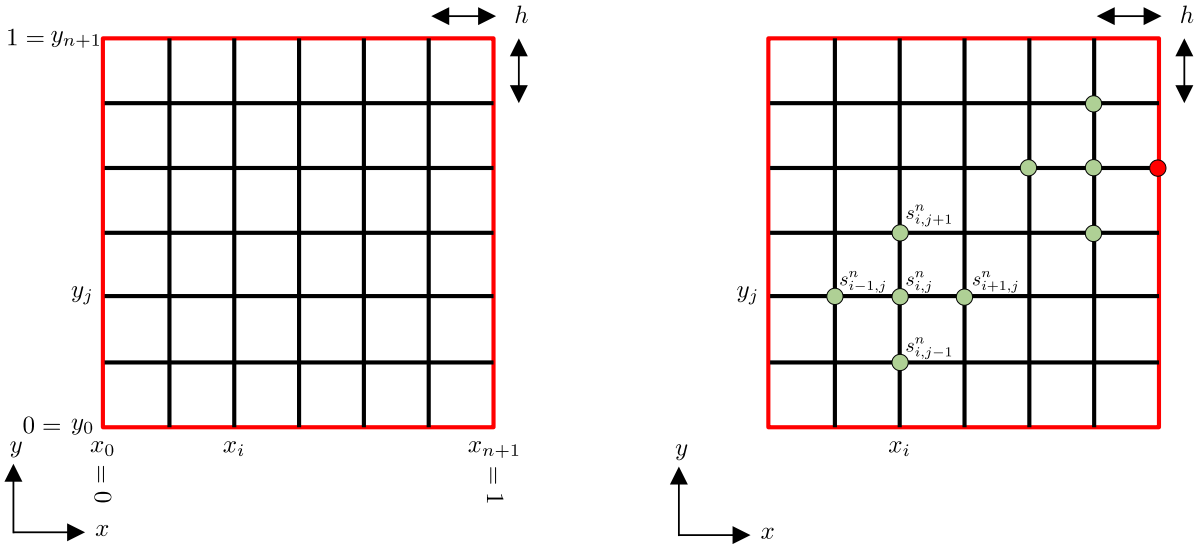


Figure 2: The left panel shows the discretization of the domain $\Omega = [0,1]^2$. The right panel shows.

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 Moodle 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 Euler cluster

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

```
[user@eu-login-39 ~]$ module load gcc
[user@eu-login-39 ~]$ make
```

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

```
[user@eu-login-39 ]$ srun --time=01:00:00 --constraint=EPYC_7763 --pty bash
srun: job 51636123 queued and waiting for resources
srun: job 51636123 has been allocated resources
[user@eu-a2p-516 ]$ ./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@eu-a2p-516 ]$ ./main 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 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.160895 seconds
1513 conjugate gradient iterations, at rate of 9403.68 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

The mini-app produces `output.bov/bin` that contain the population concentration at final time, which can be visualized with

```
[user@eu-login-39 ]$ module load python
[user@eu-login-39 ]$ python plot.py
```

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 [40 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. [20 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. [5 Points]

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

2.1 Welcome message in `main.cpp` and serial compilation [5 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@eu-a2p-409 ]$ export OMP_NUM_THREADS=4
[user@eu-a2p-409 ]$ ./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[2].

Linear algebra kernel [15 Points]

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

The diffusion stencil [15 Points]

Add OpenMP directives to parallelize 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?

Bitwise identical results [5 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[3].

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

2.3 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 $N_{\text{CPU}} = 1, 4, 16, 64$ 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 example: For base resolution $n = 64$, plot the time to solution for 64×64 on $N_{\text{CPU}} = 1$, 128×128 on $N_{\text{CPU}} = 4$, 256×256 on $N_{\text{CPU}} = 16$ and 512×512 on $N_{\text{CPU}} = 64$.
Hint: Keep in mind that the convergence of the nonlinear solver might depend on the resolution.

Additional notes and submission details

Submit **all the source code files** together with your used build files (e.g., `Makefile`(s)) and other scripts (e.g., batch job scripts) in an archive file (tar or zip) and summarize your results and observations for all sections by writing a detailed LaTeX report. Use the LaTeX template from the webpage and upload the report as a PDF to Moodle.

- Your submission should be a tar or zip archive, formatted like `project_number_lastname_firstname.zip/tgz`. It must 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 report with your name.
  - Follow the provided guidelines for the report.
- Submit your archive file through Moodle.

Please follow these instructions and naming conventions. Failure to comply results in additional work for the TAs, which makes the TAs sad...