![ETH logo]

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE   **2024**

Student: CARLA JUDITH LOPEZ ZURITA   Discussed with: FULL NAME

Solution for Project 3   Due date: Monday 15 April 2024, 23:59 (midnight)

In this report, we are solving Fisher's equation using the finite difference method. The equation is given by:

$$\frac{\partial s}{\partial t} = D\nabla^2 u + Rs(1-s) \tag{1}$$

where s is the concentration of a species, D is the diffusion coefficient, and R is the reaction rate. The equation is solved on a 2D grid with Dirichlet boundary conditions,

$$s(x,y,t) = 0.1 \quad \text{for} \quad x = 0, x = 1, y = 0, y = 1 \tag{2}$$

and initial conditions is a circle of radius 1/8 at the lower left quadrant the domain.

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

The first part of the project consists in implementing the linear algebra functions and the stencil operators. The linear algebra functions are implemented in the `linalg.cpp` file. I implemented all the functions using simple loops. Similarly, the stencil operators were implemented in the `operators.cpp` file. Writing the interior grid points was very simple since we already had the implementations for the boundary conditions.

Afterwards, I ran the code following the indications on the project description and I got the following results in the terminal

```
================================================================================
                         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 0.203392 seconds
1513 conjugate gradient iterations, at rate of 7438.82 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 1, 128, 100, 1513, 300, 0.203392 ###
Goodbye!
```

Which are the expected results, even if a bit slower than the results included in the original document, but this is just one run and could be attributed to the randomness of run-time of the processes. Additionally, I got the following content for the `output.bin` file,

```
TIME: 0.005
DATA_FILE: output.bin
DATA_SIZE: 128 128 1
DATA_FORMAT: DOUBLE
VARIABLE: phi
DATA_ENDIAN: LITTLE
CENTERING: nodal
BRICK_ORIGIN: 0. 0. 0.
BRICK_SIZE: 1 1 1.0
```
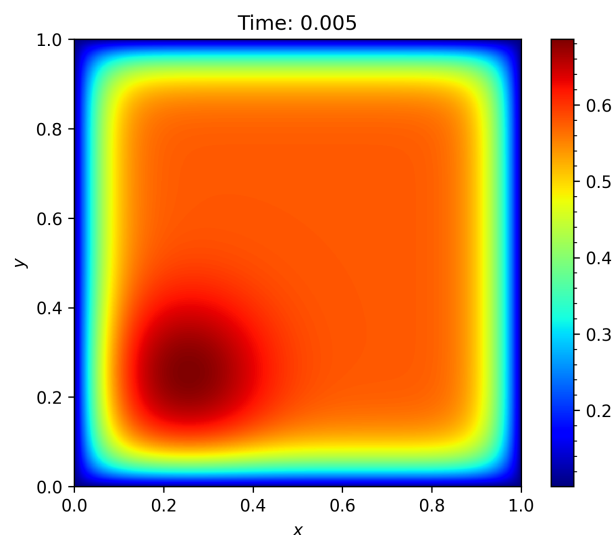
which produced Fig. 1.

Figure 1: The population concentration at time t = 0.005.

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

This section is dedicated to parallelizing the code using OpenMP.

Welcome message and serial compilation

We want to have different welcome messages for the serial and parallel versions. This was done using the macro _OPENMP which is defined when OpenMP is enabled. The code is shown below.

```
#ifdef _OPENMP
#include "omp.h" // This line won't add the library if you don't compile with -
    fopenmp option.
#endif

...

#ifdef _OPENMP
    std::cout << "version :: C++ OpenMP" << std::endl;
#pragma omp parallel
    {
        threads = omp_get_num_threads();
    }
    std::cout << "threads :: " << threads << std::endl;
#else
    std::cout << "version :: C++ Serial" << std::endl;
#endif
```

This seems to work as expected. The code is compiled with the `-fopenmp` flag to enable OpenMP.

Linear algebra kernel

Afterwards, the first step to parallelize the code was to work on the linear algebra functions. I used simple `omp parallel for` directive to parallelize the loops.

The diffusion stencil

Then, I parallelized the stencil operators. I used the same directive as before.

Bitwise identical results

I converted the bin files to hex files to be able to see clearly if the parallel implementation was producing bitwise identical results. The reults was that they were not identical.
    TODO: Discuss

Strong scaling

Next, we want to analyze the performance of the code. We will analyze the strong and weak scaling of the code. I repeated the runs 100 times for the serial and 50 for the parallel versions to get better timing results. It was a bit challenging to be able to determine the run time for certain using only one run, since there seemed to be a lot of variability in the run times. The codes to compile and run the experiments are included in the bash scripts accompanying this report. We can see the distribution of the run times in Fig. 2. Since the run times are so variable, I decided to use the median of the run times to determine the run time for the strong and weak scaling experiments. The results are shown in Figs. 3

Weak scaling

For the weak scaling, I ran the code for the parallel verison 100 times for each combination of threads and grid size. The results are shown in Fig. 4. I followed the same procedure as for the strong scaling to determine the run time.
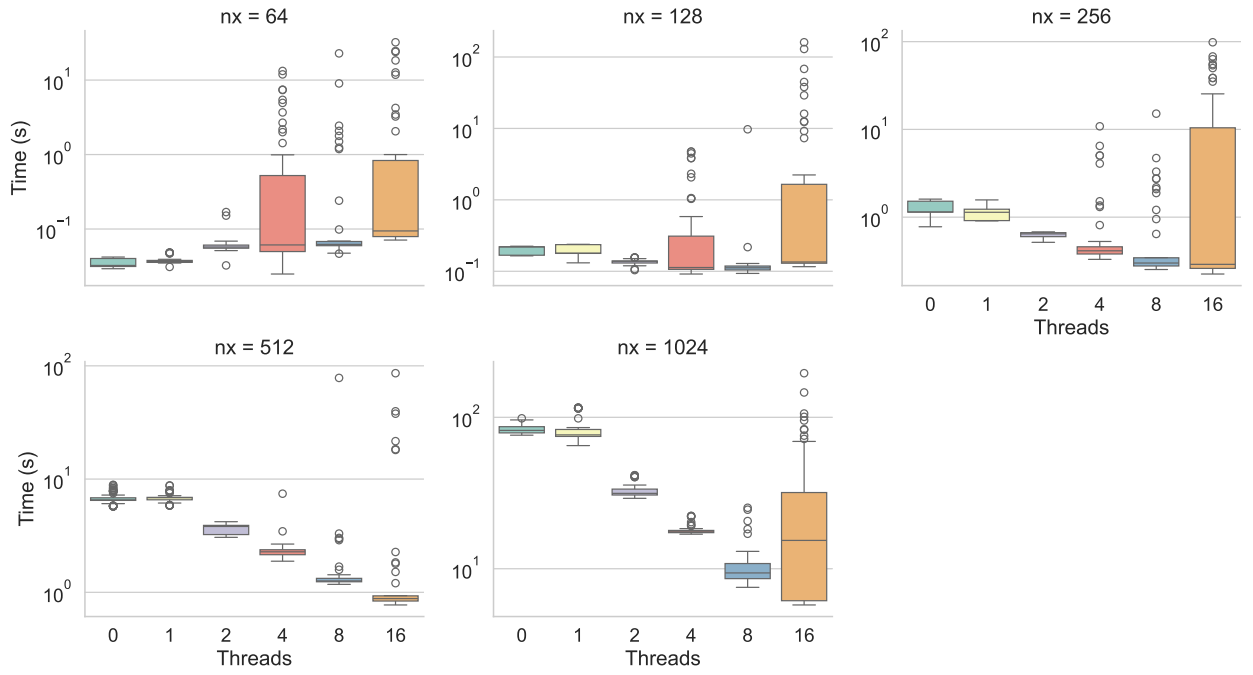
Figure 2: Time taken for the serial and parallel versions of the code; zero threads indicates serial version.
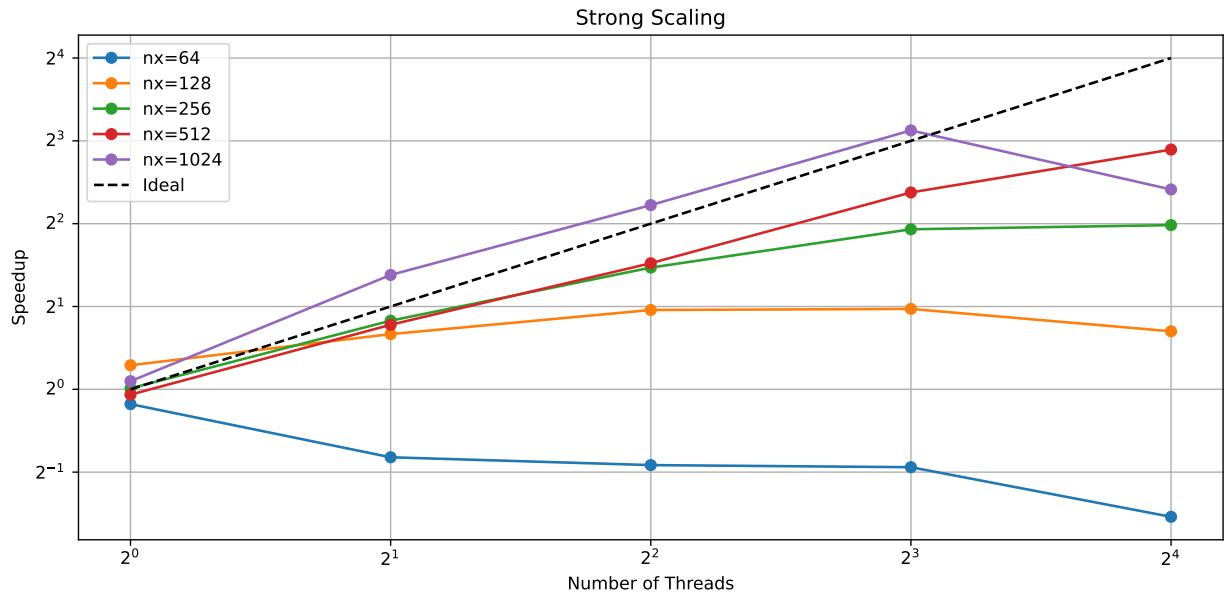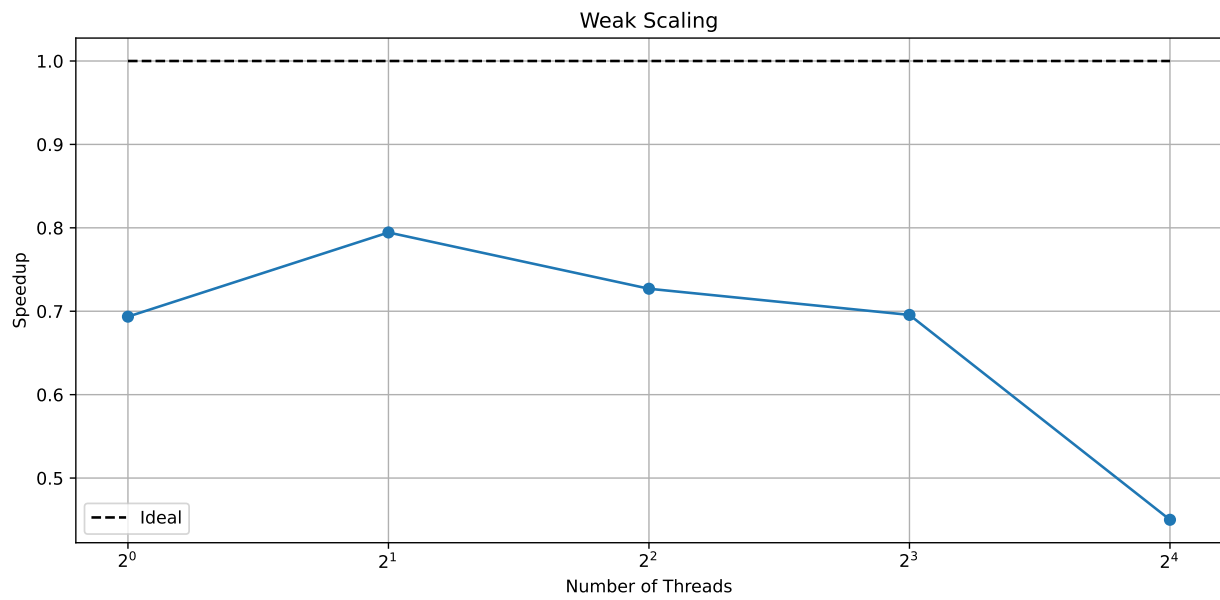

Figure 3: Strong scaling of the code.

Figure 4: Weak scaling of the code.