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

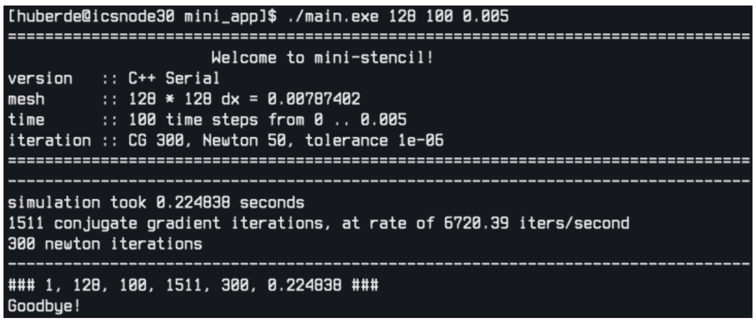
This project will introduce you a parallel space solution of a nonlinear PDE using OpenMP.

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.

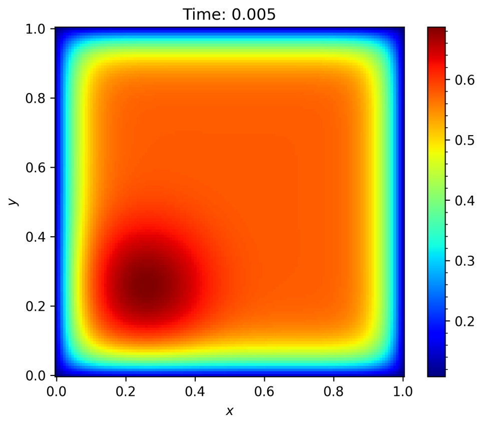
$$f_{i,j}^k = \left[-(4+\alpha)s_{i,j}^k + s_{i-1,j}^k + s_{i+1,j}^k + s_{i,j-1}^k + s_{i,j+1}^k + \beta s_{i,j}^k \left(1 - s_{i,j}^k\right)\right] + \alpha s_{i,j}^{k-1} \tag{1}$$

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]

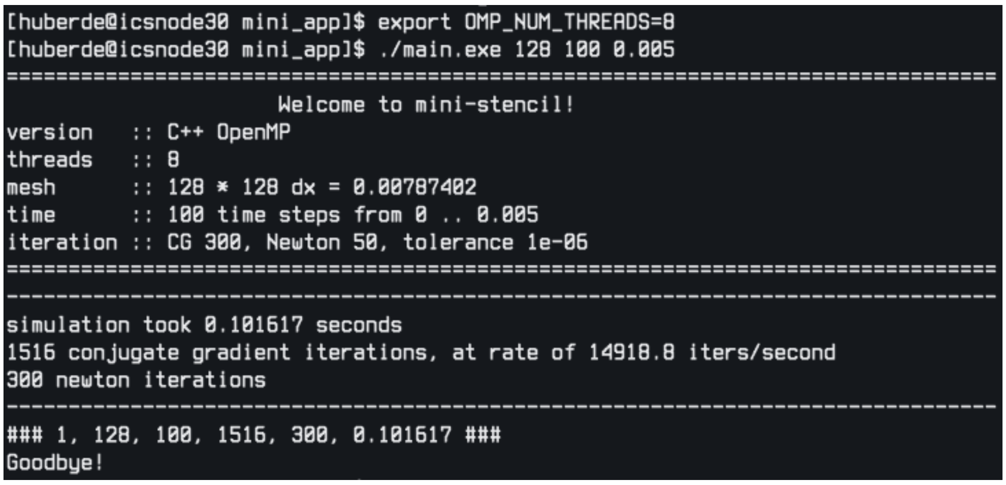
2.1. Implementation

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.

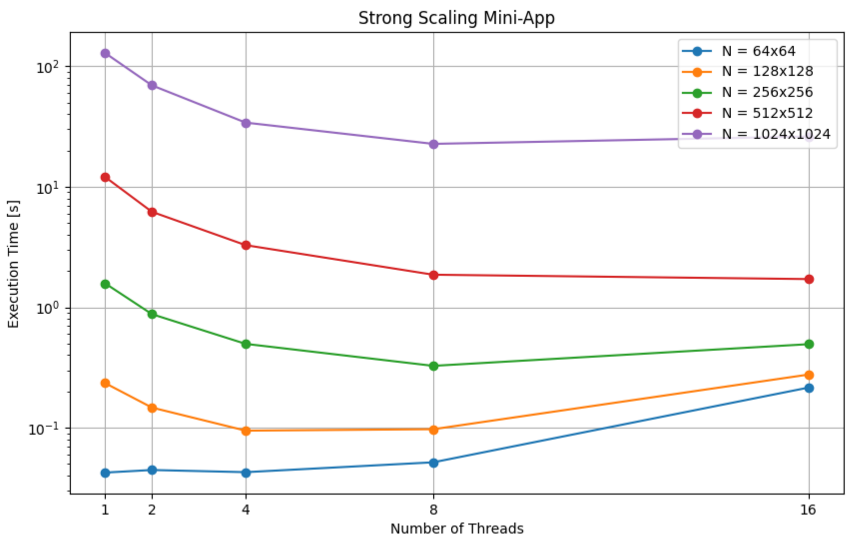
2.4. Strong & Weak Scaling Analysis

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.

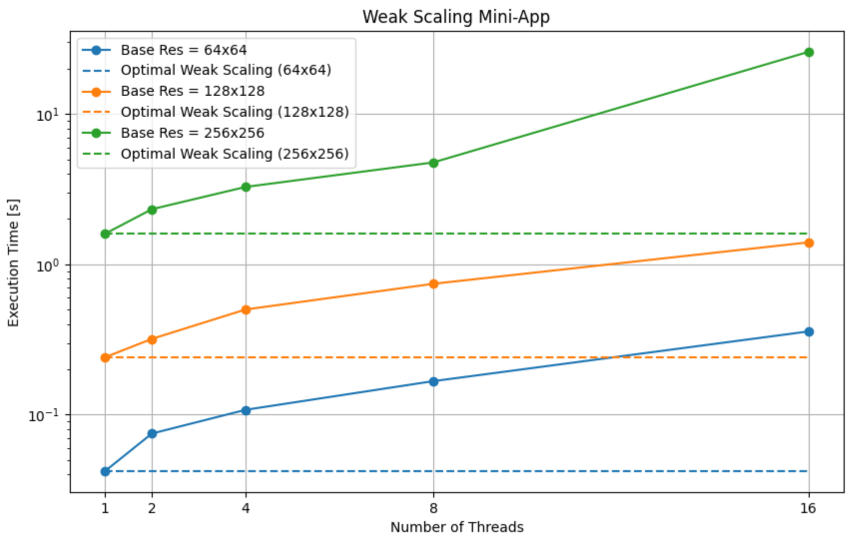
In the weak scaling scenario, when increasing the number of threads, the workload per thread stays constant. In the optimal scenario the execution time stays constant, when scaling up the number of threads and resolution. This resolution is adjusted by the following formula

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

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).