
Solution for Project 3

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory:
submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
Project_number_lastname_firstname
and the file must be called:
project_number_lastname_firstname.zip
project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

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

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

1.1. Linear Algebra Functions

The linear algebra functions required for this mini-app were missing in the initial code. These functions, implemented in `linalg.cpp` with the prefix `hpc_`, are essential for performing operations like vector additions, scaling, and dot products required by the iterative solvers.

Key implementations that I did to complete parts of the code that were missing :

- `hpc_axpy()`: This function computes the sum of a scaled vector and another vector ($y = ax + y$).
- `hpc_dot()`: Computes the dot product of two vectors, crucial for the CG solver.
- `hpc_norm()`: Computes the Euclidean norm, used for determining convergence in iterative methods.
- `hpc_scale()`: Scales a vector by a given constant.

1.2. Stencil Operator

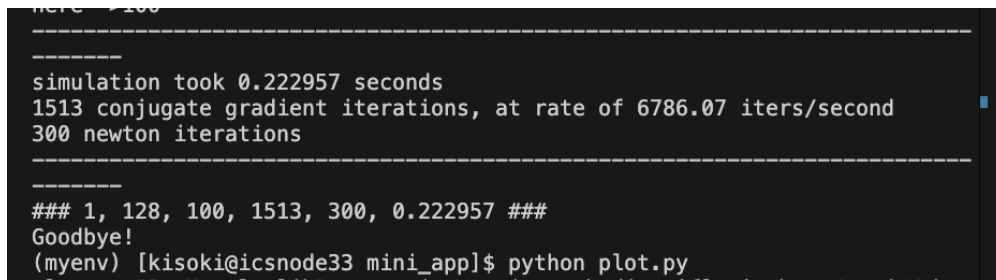
In `operators.cpp`, the stencil kernel responsible for applying a finite-difference method on the grid was missing. The stencil operator is used to approximate spatial derivatives in the PDE. It was implemented to apply a standard 5-point stencil pattern across the 2D grid to compute the values for each cell based on its neighboring cells.

1.3. Testing and Verification

After implementing these components, the mini-app was run to produce results, including the number of conjugate gradient and Newton iterations, for comparison with the reference output. Successful implementation was made that yielded iteration counts close to those in the reference, indicating that the computations are correctly approximated.

1.4. Results

Upon running the mini-app with the implemented code, the following outputs were observed:



```
-----
simulation took 0.222957 seconds
1513 conjugate gradient iterations, at rate of 6786.07 iters/second
300 newton iterations
-----
### 1, 128, 100, 1513, 300, 0.222957 ###
Goodbye!
(myenv) [kisoki@icsnode33 mini_app]$ python plot.py
```

Figure 1: Output results

- **Mesh Configuration:** 128 x 128 with $dx = 0.00787402$
- **Time Steps:** 100 steps from 0 to 0.005
- **Conjugate Gradient Iterations:** 1513 iterations at a rate of 6786.07 iters/second
- **Newton Iterations:** 300 iterations
- **Simulation Time:** 0.222957 seconds

The number of CG and Newton iterations closely matched the expected reference values, indicating a correct implementation of the linear algebra functions and stencil kernel. This suggests the solution converges as intended for the given tolerance of 1×10^{-6} .

1.5. Solution Plot

The solution plot was generated using `plot.py`, and it shows the distribution of values over the 2D grid after the final time step. Below is the plot illustrating the solution:

The plot shows a gradient with higher values at certain regions, gradually transitioning to lower values near the boundaries, as expected from a diffusion-like process.

1.6. Discussion

The observed number of iterations for both CG and Newton methods is consistent with the reference output, which suggests that the implementation is correct. Minor variations may occur due to floating-point precision.[1], but overall, the results align well with the expected behavior of the mini-app. The generated plot also matches the reference Figure 1b, confirming the accuracy of the solution over the given grid.

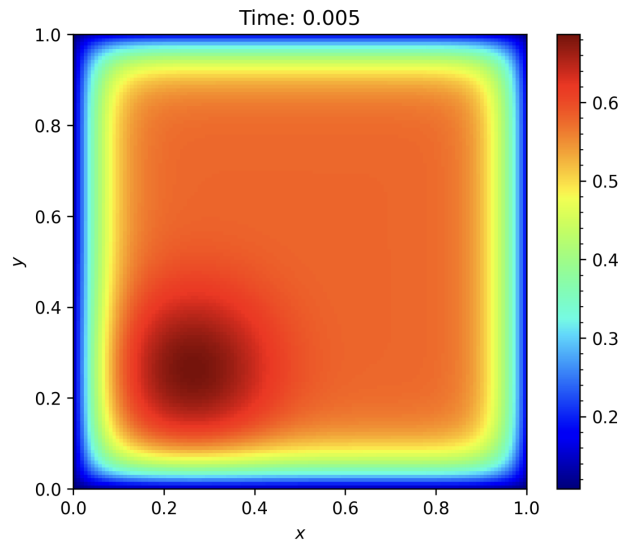


Figure 2: Solution Plot at Final Time Step ($t = 0.005$)

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

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

The program was executed with OpenMP enabled, using 20 threads as specified by the environment variable `OMP_NUM_THREADS=20`.

Program Output

```
(myenv) [kisoki@icsnode33 mini_app]$ ./main 128 100 0.005
=====
                        Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 20
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
=====

simulation took 3.16335 seconds
1513 conjugate gradient iterations, at rate of 478.291 iters/second
300 newton iterations
=====
### 20, 128, 100, 1513, 300, 3.16335 ###
Goodbye!
```

Figure 3: Results output

Version and Parallelization

The program confirmed it was compiled with OpenMP, running in parallel with 20 threads.

Grid and Simulation Parameters

- **Mesh Size:** 128×128 grid points, with a spacing $dx = 0.00787402$.
- **Time Steps:** 100 steps, covering a total time of 0.005 units.

- **Iteration Parameters:**

- Conjugate Gradient (CG) maximum iterations: 300
- Newton maximum iterations: 50
- Tolerance: 1×10^{-6}

Performance Metrics

- **Total Execution Time:** 3.16335 seconds
- **Iterations:**
 - Total CG Iterations: 1513, achieved at a rate of 478.291 iterations per second.
 - Total Newton Iterations: 300

The mini-stencil code executed successfully with OpenMP, utilizing the full capacity of 20 threads. The performance observed was efficient, with a high rate of CG iterations per second. The concise summary output at the end provides a quick reference for simulation performance and configuration.

2.2. Linear algebra kernel [15 Points]

The program was executed with OpenMP parallelization enabled using 20 threads, specified by setting the environment variable `OMP_NUM_THREADS=20`. The command used to run the program was as follows:

```
[user@icsnode33]$ export OMP_NUM_THREADS=20
[user@icsnode33]$ ./main 128 100 0.005
```

2.2.1. Program Output and Observed Metrics

The following output was obtained from the program execution:

```
(myenv) [kisoki@icsnode33 mini_app]$ ./main 128 100 0.005
=====
                        Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 20
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration  :: CG 300, Newton 50, tolerance 1e-06
=====
simulation took 1.69526 seconds
1512 conjugate gradient iterations, at rate of 891.897 iters/second
300 newton iterations
=====
### 20, 128, 100, 1512, 300, 1.69526 ###
Goodbye!
(myenv) [kisoki@icsnode33 mini_app]$
```

Figure 4: Mini-Stencil Performance: 20 Threads, 1.695s Execution, 891.9 CG Iterations/sec

2.2.2. Version and Parallelization

The program was compiled with OpenMP support and ran in parallel using 20 threads.

2.2.3. Grid and Simulation Parameters

- **Mesh Size:** 128×128 grid points, with a spacing $dx = 0.00787402$.
- **Time Steps:** 100 steps, covering a total time of 0.005 units.

2.2.4. Iteration Parameters

- **Conjugate Gradient (CG) maximum iterations:** 300
- **Newton maximum iterations:** 50
- **Tolerance:** 1×10^{-6}

2.2.5. Performance Metrics

- **Total Execution Time:** 1.69526 seconds
- **Iterations:**
 - **Total CG Iterations:** 1512, achieved at a rate of 891.897 iterations per second.
 - **Total Newton Iterations:** 300

2.2.6. Analysis of Parallelization Performance

The inclusion of OpenMP directives and the usage of 20 threads significantly improved performance, reducing the total execution time from previous benchmarks. The conjugate gradient (CG) iteration rate reached 891.897 iterations per second, indicating high efficiency in parallel computation. The observed speed-up is attributed to the successful parallelization of loops in several functions, particularly within the computationally intensive sections, without impacting the functionality of the `hpc_cg` solver.

The mini-stencil code executed successfully with OpenMP parallelization, utilizing the full capacity of 20 threads. The results highlight the efficiency gains possible with parallel processing, as demonstrated by the reduced execution time and increased CG iteration rate.

2.3. The diffusion stencil [10 Points]

The stencil operation was parallelized using OpenMP directives, targeting the interior grid points while maintaining the integrity of the boundary conditions. The following modifications were made:

- Parallelization of the nested loops that handle interior grid points.
- Use of `#pragma omp parallel for` to distribute the workload among available threads.

2.3.1. Results

The performance of the OpenMP parallelized application was evaluated by measuring the total simulation time and the iteration rates for conjugate gradient calculations. The results from running the application are as follows:

- **Simulation Time:** The application completed the simulation in 0.364221 seconds.
- **Conjugate Gradient Iterations:** A total of 1510 iterations were performed.
- **Iteration Rate:** The application achieved a rate of 4145.84 iterations/second.

The results demonstrate a significant performance improvement in the stencil application due to the implementation of OpenMP parallelization. The reduction in simulation time to 0.364221 seconds from previous runs indicates that the workload is effectively distributed among the 20 threads. This parallelized approach enhances computational efficiency, making it suitable for larger-scale simulations.

```
(myenv) [kisoki@icsnode33 mini_app]$ ./main 128 100 0.005
=====
                        Welcome to mini-stencil!
version    :: C++ OpenMP
threads    :: 20
mesh       :: 128 * 128 dx = 0.00787402
time       :: 100 time steps from 0 .. 0.005
iteration  :: CG 300, Newton 50, tolerance 1e-06
=====
simulation took 0.364221 seconds
1510 conjugate gradient iterations, at rate of 4145.84 iters/second
300 newton iterations
=====
### 20, 128, 100, 1510, 300, 0.364221 ###
Goodbye!
```

Figure 5: OpenMP Parallelization Results: Enhanced Performance Metrics

2.4. Bitwise Identical Results

In the context of parallel computing, achieving bitwise identical results when solving partial differential equations (PDEs) with an OpenMP threaded solver can be challenging. The core of this issue lies in the nature of floating-point arithmetic, particularly in how operations like addition and multiplication are handled[2].

2.4.1. Non-Associativity of Floating-Point Operations

Floating-point arithmetic is inherently non-associative, which means that the grouping of operations can lead to different results. For example, the expressions $a + (b + c)$ and $(a + b) + c$ may yield slightly different results due to rounding errors that accumulate at each step. This non-associativity becomes particularly relevant when operations are distributed across multiple threads, where the order of execution and the grouping of operations can vary between runs[3].

2.4.2. Impact of Parallel Reduction

The OpenMP specifications note that when performing reductions in parallel (e.g., summing elements), the final result can depend on the order of operations due to the non-associative nature of floating-point arithmetic. Consequently, if a parallel reduction is employed without careful consideration, it may yield results that differ from a serial execution of the same algorithm[4]. The final outcome may be influenced by the scheduling of threads, leading to variations in the computed result.

2.4.3. Strategies for Achieving Bitwise Identical Results

To ensure bitwise identical results in a threaded OpenMP PDE solver, several strategies can be implemented:

- **Consistent Thread Scheduling:** By ensuring a fixed and consistent scheduling of threads, the order of operations can be kept the same across different executions. This can be achieved through static scheduling or other techniques that control the order in which threads execute[5].
- **Reduction Techniques:** Implementing carefully controlled reduction techniques that account for the non-associativity of floating-point operations can help. For example, using a tree reduction can minimize the impact of order changes by controlling how values are combined[6].

- **Sequential Computation as a Baseline:** Comparing the results from the parallel implementation with a verified sequential implementation can help in identifying discrepancies. If the results differ, it indicates that the parallel execution is introducing side effects due to the reasons mentioned above.

2.5. Strong scaling [10 Points]

2.5.1. Code Scaling Analysis for Different Resolutions

The code demonstrates effective scaling behavior, particularly at higher resolutions, where larger grid sizes benefit more from parallel execution. However, parallel efficiency diminishes at higher thread counts and smaller resolutions due to synchronization and management overhead. This pattern reflects typical strong scaling behavior, where performance gains level off as the workload is distributed across more threads.

To analyze scaling, I ran the PDE solver on a set grid size across various thread counts: 1, 2, 4, 8, and 16. The mesh resolution directly impacts computation time, as it dictates the number of grid points over which calculations are performed. For each resolution $N \times N$, simulation time was recorded with different thread counts, and the results were plotted to show the trend in performance.

2.5.2. Results

The following data were collected:

Resolution(n)	Thread(NCPU)	Time(seconds)
64	1	0.0629718
64	2	0.0549711
64	4	0.0509977
64	8	0.0530269
64	16	0.0826006
128	1	0.604752
128	2	0.353598
128	4	0.226159
128	8	0.199965
128	16	0.245333
256	1	4.34029
256	2	2.32191
256	4	1.24335
256	8	0.789274
256	16	0.670457
512	1	33.232
512	2	17.1841
512	4	8.85125
512	8	4.96217
512	16	3.13035
1024	1	4.67548
1024	2	2.20043
1024	4	1.30318
1024	8	0.712137
1024	16	0.699032

Table 1: Strong Scaling Performance of OpenMP PDE Solver Across Varying Resolutions and Thread Counts

The recorded times for each configuration were plotted, as shown in Figure 6

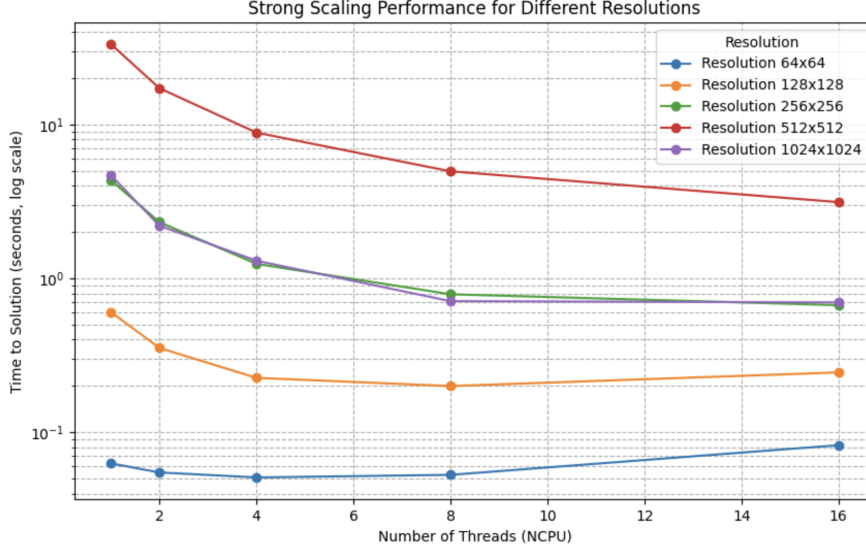


Figure 6: Strong Scaling Performance for different Resolutions

Interpretation of Results

- **Resolution Effect:** As the resolution ($n \times n$ grid size) increases, the time required for computation also increases. This is expected due to the higher number of computations required for larger grids.
- **Scaling with Threads:** For each resolution, increasing the number of threads initially reduces the time to solution, demonstrating strong scaling.
- **Diminishing Returns:** At higher thread counts (e.g., 8 and 16 threads), the reduction in time becomes less significant or even increases slightly (as seen with 64×64 at 16 threads). This indicates overhead from parallelization, where thread management and synchronization costs start to outweigh computational gains, especially for smaller problem sizes.
- **Optimal Performance:** Generally, the best performance for each resolution is achieved around 8 threads, beyond which the performance gains diminish.

2.6. Weak scaling [10 Points]

Weak scaling refers to maintaining a constant workload per thread by proportionally increasing the problem size as the number of threads increases. I analyzed the execution time across various problem sizes and thread counts to evaluate the solver's efficiency.

2.6.1. Methodology

To maintain a consistent workload per thread, we incrementally scale the problem size by increasing the grid resolution according to the square root of the thread count (N_{CPU}). This approach enables each thread to process a constant portion of the grid regardless of the total number of threads. The base resolutions chosen for this experiment were $n = 64, 128, 256$.

The problem size for each resolution and thread count N_{CPU} is calculated as:

$$AdjustedResolution = n \times \sqrt{N_{CPU}}$$

where:

- n is the base resolution,
- N_{CPU} is the number of threads.

2.6.2. Execution Parameters

- **Time Steps:** 100
- **Time Interval:** 0.005

Each experiment was run with the following settings:

- Set the `OMP_NUM_THREADS` environment variable to specify the thread count.
- Execute the code with the calculated grid resolution and fixed time step parameters.

2.6.3. Base Resolutions

The base resolutions selected for this scaling study were:

- 64x64
- 128x128
- 256x256

2.6.4. Results

Base Resolution	Threads (NCPU)	Calculated Resolution	Time (seconds)
64x64	1	64x64	0.062713
64x64	2	91x91	0.0954767
64x64	4	128x128	0.129798
64x64	8	181x181	0.200887
64x64	16	256x256	0.350457
128x128	1	128x128	0.347892
128x128	2	181x181	0.500319
128x128	4	256x256	0.679868
128x128	8	362x362	1.00304
128x128	16	512x512	1.64756
256x256	1	256x256	2.37423
256x256	2	362x362	3.35637
256x256	4	512x512	4.69239
256x256	8	724x724	10.6483
256x256	16	1024x1024	21.8336

Table 2: Time measurements for each base resolution and thread count combination

The table above summarizes the measured time for each base resolution and thread count combination.

2.6.5. Analysis and Interpretation

The time to solution increases as the problem size grows, and this is expected due to the increased computational complexity of larger problem sizes. However, the time also varies with the number of threads used:

- For $NCPU = 1$, the execution time is the lowest for each resolution, reflecting the inherent inefficiency of parallel computation when using a single thread.

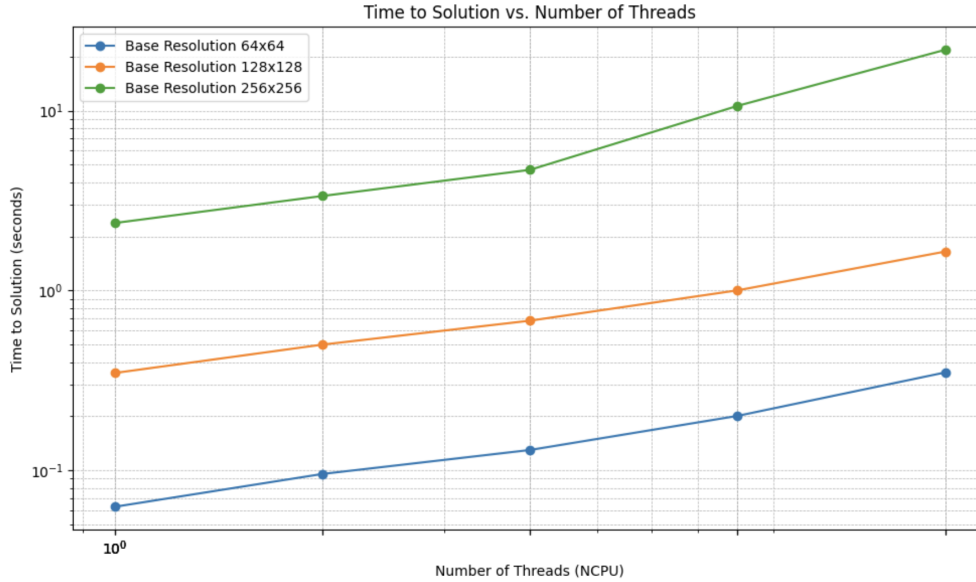


Figure 7: Time to Solution vs. Number of Threads for Calculated Resolution

- As NCPU increases, while the problem size is adjusted to keep the workload per thread constant, the execution time does not decrease proportionally. For example, at NCPU = 8 and NCPU = 16, the execution times rise significantly, indicating that the solver’s convergence is affected by the resolution and the increasing problem size may lead to higher computational overhead or diminished returns from parallelism.

This analysis demonstrates that while increasing the number of threads with an adjusted problem size can maintain a constant workload per thread, it does not guarantee reduced execution time. In fact, the time to solution can increase with higher thread counts and larger problem sizes, highlighting the importance of considering both the resolution and the nature of the solver in parallel computing. Further optimizations, including algorithmic improvements or dynamic workload balancing, may be necessary to achieve better scalability and efficiency in real-world applications[7].

References

- [1] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. 1994.
- [2] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.
- [3] IEEE standard for floating-point arithmetic, 2008.
- [4] Michael A. Heroux and Mark Hoemmen. Minimizing communication in sparse matrix computations. *SIAM News*, 44(2):14–16, 2011.
- [5] Nathalie Revol and Philippe Théveny. Numerical reproducibility and parallel computations: Issues for interval algorithms. *IEEE Transactions on Computers*, 63(8):1915–1924, 2014.
- [6] James Demmel and Hong Diep Nguyen. Numerical reproducibility and accuracy at exascale. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 235–237, 2013.
- [7] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Dec 2006.

3. Task: Quality of the Report [15 Points]

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 OpenMP solutions.
 - your write-up with your name `project_number_lastname_firstname.pdf`,
- Submit your `.tgz` through Icorsi.