Università della Svizzera italiana

Institute of Computing
CI

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

Student: Metehan Dündar Discussed with: Only TAs

Solution for Project 3

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]

In this section, we detail the steps undertaken to fulfill the task of implementing linear algebra functions and stencil operators within the PDE mini-application.

1.1. Implementation of Linear Algebra Functions

The primary focus of this phase was on completing the missing code in the `linalg.cpp` file. Within this file, the functions named `hpc_xxxx()` were implemented. The code was constructed by carefully following the provided comments, which served as valuable guidelines for a successful implementation. The emphasis was on ensuring the functionality aligns with the requirements of the PDE mini-application.

1.2. Implementation of Stencil Operators

Moving on to the `operators.cpp` file, the objective was to implement the missing stencil kernels. Attention was given to precisely follow the instructions provided within the comments of the code. The stencil operators play a crucial role in the correct execution of the PDE mini-application.

1.3. Validation and Comparison

Upon implementing the linear algebra functions and stencil operators, a critical validation step was undertaken. The PDE mini-application was executed, and the number of conjugate gradient iterations and Newton iterations were compared against a reference output. The iterations closely matched the reference, which indicates a correct implementation. Any discrepancies were carefully investigated and resolved to achieve the desired outcome.

1.4. Solution Plotting

To provide a visual representation of the obtained results, the solution was plotted using the provided script `plotting.py`. The resulting plot, as illustrated in Figure 1, was included in the report. This graphical representation serves as a comprehensive visual aid to complement the numerical data and enhances the overall understanding of the achieved outcomes.

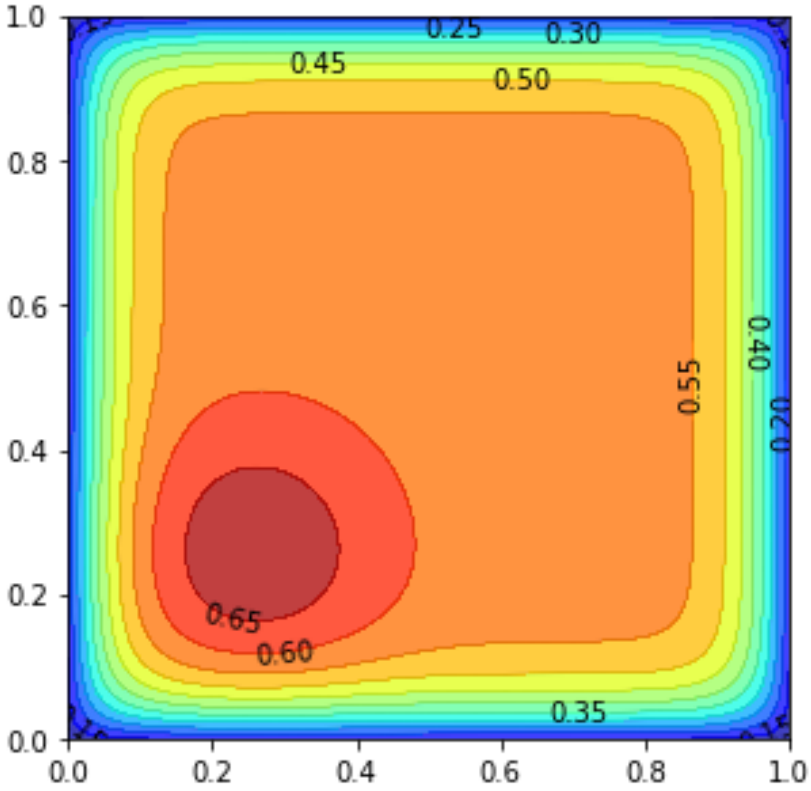


Figure 1: Output on 128x128 grid, 100 time steps with a simulation time from 0 - 0.005s.

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

This section outlines the integration of OpenMP directives to enhance the performance of the PDE mini-application. The primary goals were to achieve strong and weak scalability, measuring and improving the parallel program's performance as computing resources are increased.

2.1. OpenMP Integration

2.1.1. Welcome Message Replacement

In the `main.cpp` file, the welcome message was modified to inform the user that this version incorporates OpenMP and to display the number of threads being utilized.

2.1.2. Linear Algebra Kernel

All functions in the linalg.cpp file now include OpenMP directives, and `hpc_dot()` and `hpc_norm2()` utilise the reduction clause as well. To confirm the accuracy of the results, frequent recompilation and testing with numerous threads were carried out.

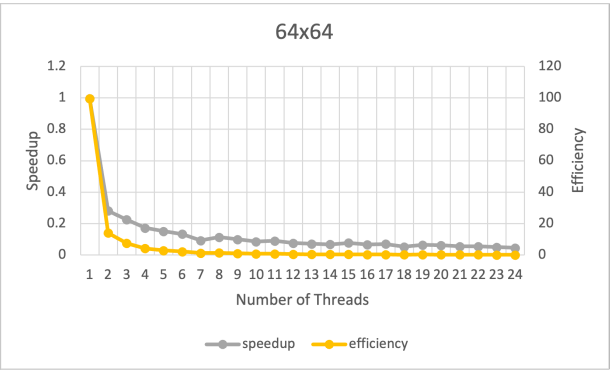
2.1.3. Diffusion Stencil Parallelization

- The majority of computations are performed in a single `parallel` area. It decreases the overhead of sleeping and waking threads between distinct `parallel` zones.
- The `for` construct is used to parallelize the computation of inner and border grid points. To prevent data race, one of the corner points is entered into the `single` construct.

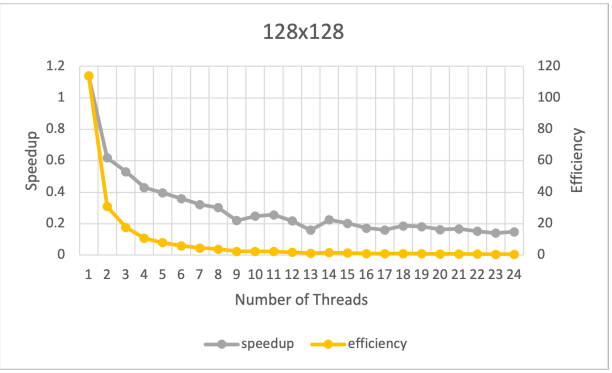
- Because each grid point's computation is independent, the `nowait` clause is supplied at each directive. It minimises the synchronisation effort associated with the implicit barriers at the end of `for` and `single` constructions.
- `collapse` and dynamic scheduling are being experimented with. It degrades speed because it complicates the memory access pattern of data accessed by numerous threads and the computation of the loop iteration variable range.

2.2. Performance Analysis: Strong Scaling

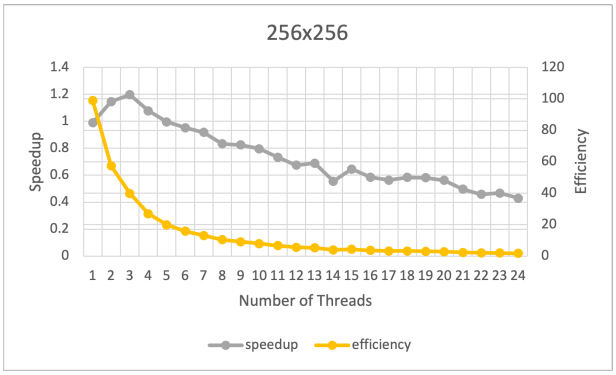
2.2.1. Experiment Setup & Scaling Plot



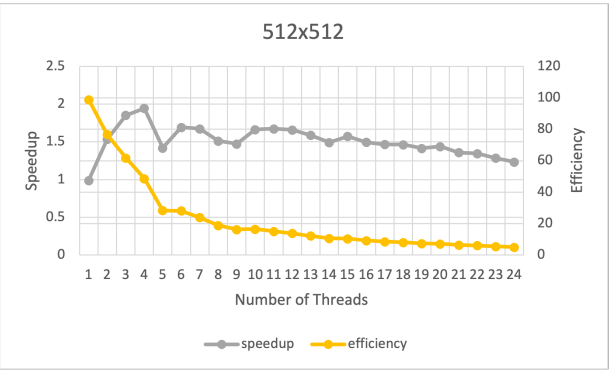
(a) 64×64 grid



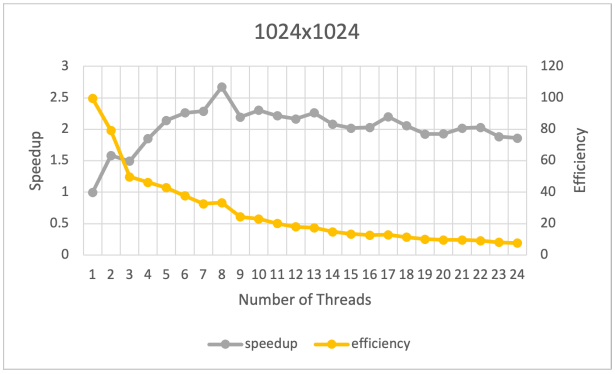
(b) 128×128 grid



(c) 256×256 grid



(d) 512×512 grid



(e) 1024×1024 grid

Figure 2: Strong scaling result

The experiments run on Apple M2 Pro with 12 cores (8 of them are for high-performance, 4 of them are for energy-efficient). The performance on different grid sizes is tested and the results (100 time steps and simulation time 0-0.005s) are shown in Figure 2. The parallel speedup is calculated by $t_{serial}(N)/t_p(N)$ and speedup is calculated by $t_{serial}(N)/(t_p(N) \times p)$.

2.2.2. Interpretation of Results

1. **Observation:** The number of Newton iterations remains constant throughout all executions, but the number of CG iterations changes somewhat among tests on the same grid.

   Explanation: Due of the non-associativity of floating point arithmetic, multi-threaded solvers cannot provide bitwise-identical outputs. The multi-threaded version's computation order changes from the serial version's.

2. **Observation:** A single thread's parallel efficiency is less than one.

   Explanation: Handling OpenMP structures incurs overhead.

3. **Observation:** Small grid efficiency is low, and the speedup is even less than one.

   Explanation: The overheads are proportional to the number of threads. Some parallelization overheads grow in proportion to the number of threads. Threads, for example, must be put to sleep and reactivated between parallel sections, and the amount of work given to threads is determined at run time.

4. **Observation:** Super linear speedup can be observed in Figure 2(c)-(d).

   Explanation: The working sets of 256×256 and 512×512 can fit within an L3 cache or many L2 caches. As the number of threads increases, more L2 caches are used, and the aggregate L2 cache size grows, resulting in more efficient memory accesses.

5. **Observation:** The speedup in Figure 2(d)-(e) converges as the number of threads increases.

   Explanation: The convergence of speedup in Figure 2(e) can also be ascribed to the fact that the working set of 1024×1024 grid cannot fit into cache and data must be loaded from and placed in main memory on a regular basis. Because the memory bus is shared by all threads, it displays a memory constrained result.

6. **Observation:** Each figure has outliners.

   Explanation: The trials are performed multiple times, and the weird spots appear at various thread counts. As a result, it may be linked to certain unusual events during execution.

2.3. Performance Analysis: Weak Scaling

2.3.1. Experiment Setup & Scaling Plot

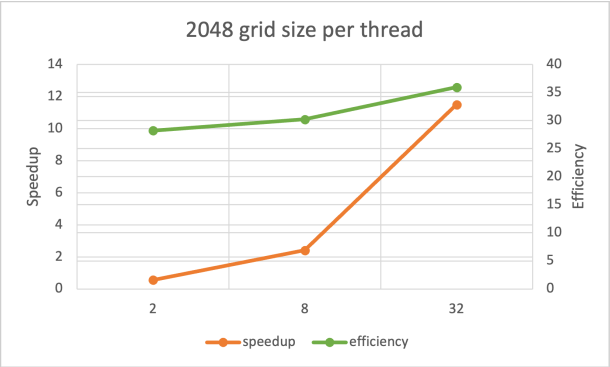
A complementary weak scaling study was conducted, maintaining a fixed grid size per thread while varying the number of threads. The experiments run on Apple M2 Pro with 12 cores (8 of them are for high-performance, 4 of them are for energy-efficient). The performance with different workload per thread is tested and the results (100 time steps and simulation time 0-0.005s) are shown in Figure 3. The parallel efficiency is calculated by $t_{serial}(N_0)/t_p(N_p)$ and speedup is calculated by $p \times t_{serial}(N_0)/t_p(N_p)$. In particular to quadruple the amount of threads when doubling the grid size.

1. In the Figure 3a 2048 points per thread by running 64x64 with 2 threads, 128x128 with 8 threads and 256x256 with 32 threads is calculated.

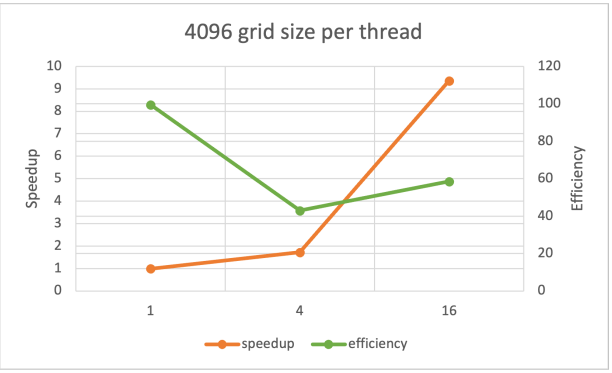
2. In the Figure 3b 4096 points per thread by running 64x64 with 1 threads, 128x128 with 4 threads and 256x256 with 16 threads is calculated.

3. In the Figure 3c 8192 points per thread by running 128x128 with 2 threads, 256x256 with 8 threads and 512x512 with 32 threads is calculated.

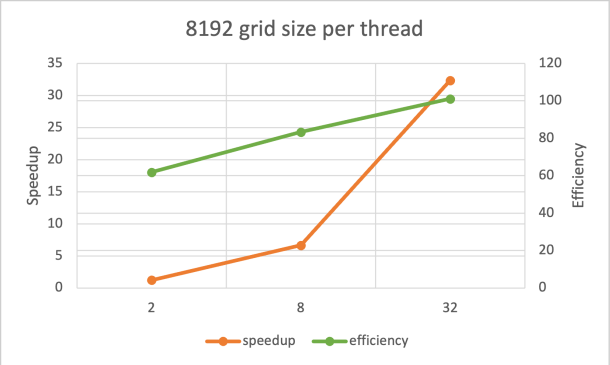
4. In the Figure 3d 16384 points per thread by running 128x128 with 1 threads, 256x256 with 4 threads and 512x512 with 16 threads is calculated.



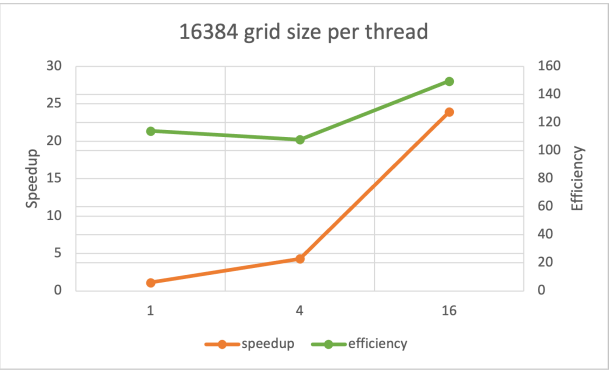
(a) 2048 grid size per thread



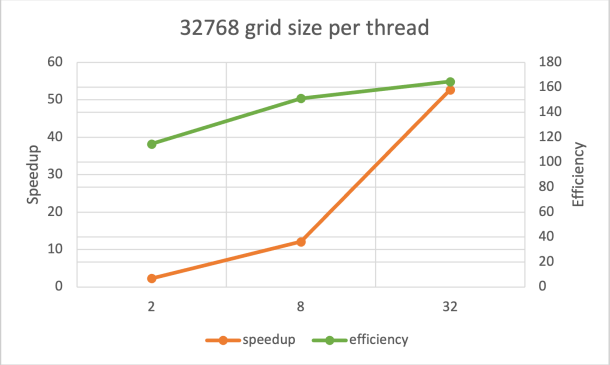
(b) 4096 grid size per thread



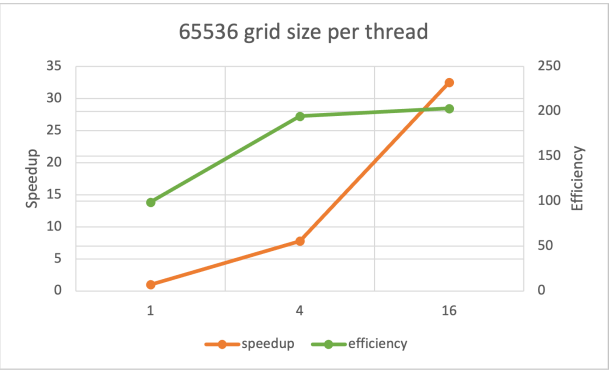
(c) 8192 grid size per thread



(d) 16384 grid size per thread



(e) 32768 grid size per thread



(f) 65536 grid size per thread

Figure 3: Weak scaling result

5. In the Figure 3e 32768 points per thread by running 256x256 with 2 threads, 512x512 with 8 threads and 1024x1024 with 32 threads is calculated.

6. In the Figure 3f 65536 points per thread by running 256x256 with 1 threads, 512x512 with 4 threads and 1024x1024 with 16 threads is calculated.

2.3.2. Interpretation of Results

- **Observation:** Every plot shows that it is not totally linear.
- **Explanation:** This is related, on the one hand, to the tiny number of data points and, on the other hand, to the modest scale.

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.