Università
della
Svizzera
italiana

Institute of Computing CI

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

Student: Paolo Deidda Discussed with: FULL NAME

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 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.

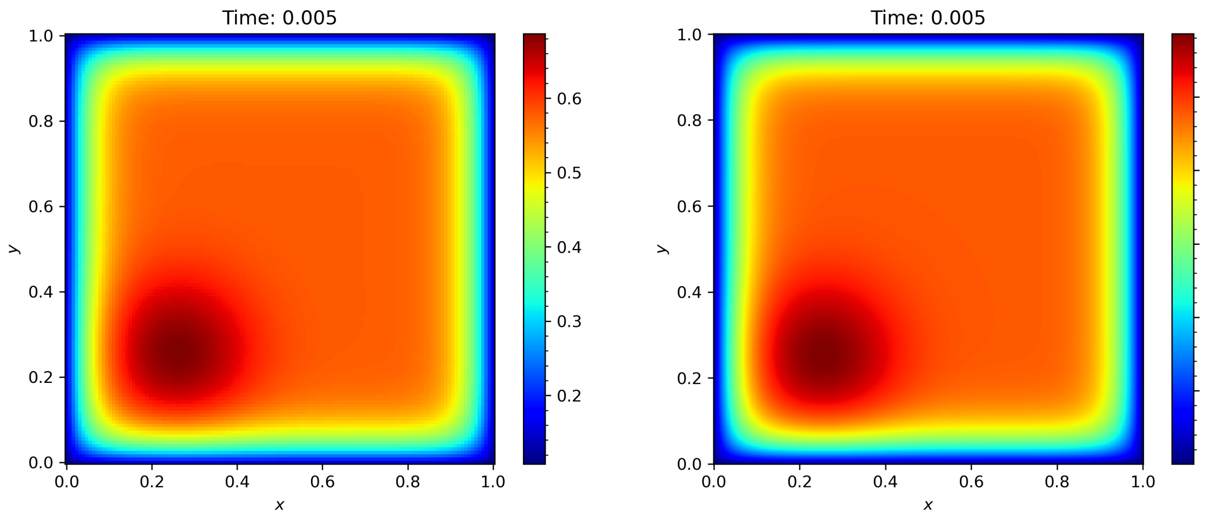
This project will introduce you a parallel space solution of a nonlinear PDE using OpenMP.

Contents

1. Task 1: Implementing the Linear Algebra Functions and the Stencil Operators [35 Points]

The completion of the functions was straightforward, since their structure followed standard BLAS 1 patterns and a classical 5–point stencil layout. The code can be found in the linalg.cpp and operators.cpp files.

Figure 1 shows the simulation results for two different grid resolutions: a lower resolution run ($n = 128$) and a high–resolution run ($n = 1024$), respectively.



(a) Concentration at final time for a low resolution grid ($n = 128$).

(b) Concentration at final time for a high resolution grid ($n = 1024$).

Figure 1: Simulation results for two different grid resolutions.

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 welcome message in main.cpp was updated to include the OpenMP version and the number of threads detected at runtime. To get the current number of threads I used OpenMP's function:

```
    int threads = omp_get_max_threads();
```

Listing 1: Welcome message update in main.cpp

This correctly displays the OpenMP version and the number of threads in the standard output:

```
==========================================
Running with N=64, NT=100, TF=0.005, threads=4
================================================================================
                      Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 4
mesh      :: 64 * 64 dx = 0.015873
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
```

Listing 2: Output example with OpenMP version and number of threads

2.2. Linear algebra kernel [15 Points]

I added OpenMP directives to all BLAS 1 functions (except `hpc_cg`). I mostly used `#pragma omp parallel for` to parallelize vector operations, and `reduction` clauses so that the threads create private copies of the reduction variables and only the final result is combined reducing critical merge operation that must be done atomically. The implementation can be found in the linalg.cpp and operators.cpp files.

```
// computes the inner product of x and y
// x and y are vectors on length N
double hpc_dot(Field const& x, Field const& y, const int N) {
    double result = 0;

    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < N; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

Listing 3: Example from dot product implementation in linalg.cpp

2.3. Diffusion stencil [10 Points]

I used the OpenMP special keyword `collapse` to parallelize the nested loops so that the two loops are treated as a single loop with a larger iteration space. This should better distribute the workload among threads, especially for larger grids.

```
    // the interior grid points
    #pragma omp parallel for collapse(2)
    for (int j=1; j < jend; j++) {
        for (int i=1; i < iend; i++) {
            f(i,j) = -(4. + alpha) * s_new(i,j)
                   + s_new(i-1,j) + s_new(i+1,j)
                   + s_new(i,j-1) + s_new(i,j+1)
                   + alpha * s_old(i,j)
                   + beta * s_new(i,j) * (1.0 - s_new(i,j));
        }
    }
```

Listing 4: Example from diffusion stencil implementation in operators.cpp

Sections in the boundary loops (North, South, East, West) remain sequential: they are anyway small, inexpensive, and include edge conditions that could introduce thread competition if parallelised.

2.4. Bitwise identical results [3 Points]

Definition. Bitwise-identical result would mean that every bit of the final output (each `double` or `float`) is exactly the same between the serial and the parallel execution. So **not** "almost the same" (e.g., `1.00000000` vs. `1.00000001`), but rather *identical in memory.* Getting to this level of reproducibility is really tough and for most numerical parallel programs, it's pretty much impossible.

Why OpenMP cannot guarantee bitwise-identical results.

1. **Floating-point non-associativity.** In floating-point arithmetic, addition and multiplication are not associative:

$$(a + b) + c \neq a + (b + c)$$

due to rounding errors in intermediate operations. For example:

$$(10^{16} + 1) - 10^{16} = 0, \qquad 10^{16} + (1 - 10^{16}) = 1$$

You can get different outcomes from the same numbers based on how they're evaluated. When using OpenMP to run loops in parallel, and multiple threads are doing their own sums, the way those sums get combined isn't always the same. This can lead to tiny variations at the bit level in the final result.

2. **Reductions in OpenMP.** When using a reduction clause such as:

```
    #pragma omp parallel for reduction(+:sum)
```

   OpenMP makes a separate copy of `sum` for each thread, and then combines (or reduces) all those private results but the order in which it does that isn't always the same. As a result, I ended up with slightly different outputs for each run.

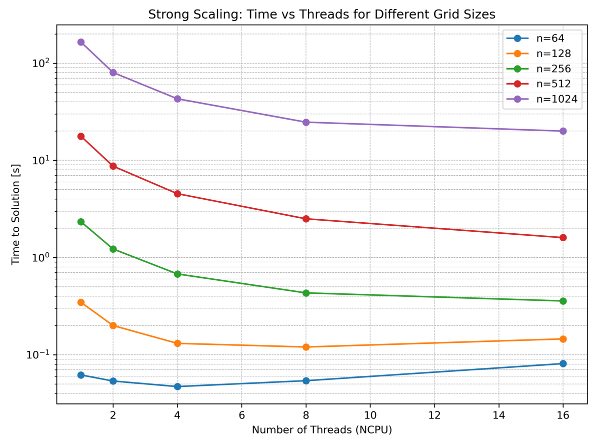
3. **Dynamic scheduling.** When we use directives like `schedule(dynamic)` or `schedule(guided)`, the way iterations get assigned to threads can differ from one run to the next. This change messes with the order in which values are accumulated.

In theory, it is possible to get bitwise-identical results, but doing so means giving up a lot of the benefits that come with parallel processing. To make this happen, we'd need to:

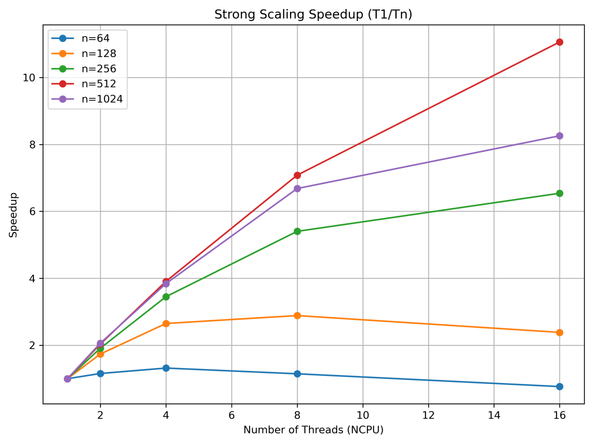
- use `schedule(static)` with deterministic workload distribution;
- avoid all reductions or enforce a fixed reduction tree to preserve a consistent operation order;
- force threads to reproduce the exact same operation sequence as the serial version.

In practice, these limitations would really hurt scalability and complicate the code more than necessary.

2.5. Strong scaling [10 Points]



(a) Strong scaling: Time vs. Number of Threads

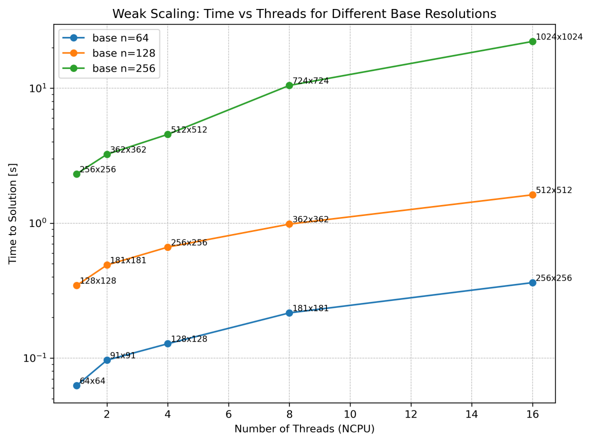


(b) Strong scaling: Speedup vs. Number of Threads

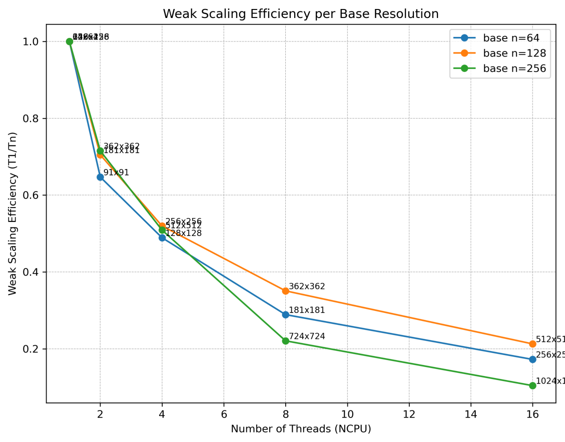
Figure 2: Strong scaling results for the OpenMP parallelization of the PDE mini-app.

The results show that as we increase the number of threads, we see almost a linear boost in speed, apart from low resolution runs (n=64 blue and n=128 orange). OpenMP effectively utilizes multiple threads to speed up computations, however, for lower resolutions, the threads spend more time coordinating rather than computing.

2.6. Weak scaling [10 Points]



(a) Weak scaling: Time vs. Number of Threads



(b) Weak scaling: Efficiency vs. Number of Threads

Figure 3: Weak scaling behavior for the OpenMP implementation.

In my weak scaling tests, I noticed that as we added more threads, execution time increased almost linearly. The idea is that if we keep the workload per thread the same, the time it takes to get a solution should stay constant. We're seeing performance drop mainly because of synchronization costs and general overhead, also due to memory bandwidth getting maxed out. Since all threads are tapping into the same memory, just adding more threads doesn't boost the available memory bandwidth, so total runtime ends up growing linearly with the number of threads.

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