Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Student: Benedict Armstrong Discussed with: Tristan Gabl

Solution for Project 3

Due date: Monday 15 April 2024, 23:59 (midnight)

1. Implementing the linear algebra functions and the stencil operators

1.1. Linalg functions

Implementing the eight linalg function outlined in `linalg.cpp` was relatively straighforward. Each followed a similar pattern of component wise iteration over the input array(s) and then performing the required operation. An example of the `copy` function is shown in Listing 1.

```
for (int i = 0; i < N; i++)
{
    y[i] = x[i];
}
```

Listing 1: Linalg copy function

1.2. Stencil operators

The next task was Implementing the stencil operator. Listing 2 shows how we calculate the value for each grid cell.

```
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)
        + beta * s_new(i, j) * (1.0 - s_new(i, j))
        + alpha * s_old(i, j);
```

Listing 2: Stencil operator

1.3. Plotting the results

Finally we can plot the results with the following parameters:

- $nx = ny = 128$
- $nt = 100$
- $t = 0.005$

The output of the serial version is shown in Listing 3.
The results are shown in Figure 1.

```
$ ./build/main 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 0.15112 seconds
1514 conjugate gradient iterations, at rate of 10018.5 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 1, 128, 100, 1514, 300, 0.15112 ###
Goodbye!
```

Listing 3: Running the serial version of the mini-app

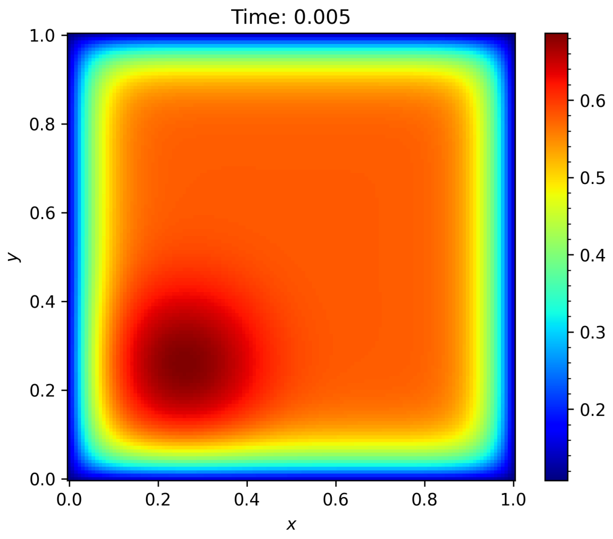


Figure 1: Results of the nonlinear PDE mini-app

2. Adding OpenMP to the nonlinear PDE mini-app

First I reconfigured the project welcome message. If `_OPENMP` is defined, we use `omp_get_max_threads()` to get the number of threads. The code welcome message is shown in Listing 4.

```
#ifdef _OPENMP
    std::cout << "version   :: C++ OpenMP" << std::endl;
    std::cout << "threads   :: " << threads << std::endl;
#else
    std::cout << "version   :: C++ Serial" << std::endl;
#endif
```

Listing 4: New OpenMP welcome message

Next I added parallelised versions of the linalg functions. For most of the functions, I simply added the *#pragma omp parallel for* directive before the loop. For the dot product and the norm functions, I used the `reduction` clause to ensure the correct result. An example of the copy function is shown in Listing 5.

```
#pragma omp parallel for shared(y, x)
    for (int i = 0; i < N; i++)
    {
        y[i] = x[i];
    }
```

Listing 5: OpenMP copy function

Finally, I added the *#pragma omp parallel for collapse(2)* directive to the stencil operator to parallelise the nested loops. The updated stencil operator is shown in Listing 6.

```
#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)
                    + beta * s_new(i, j) * (1.0 - s_new(i, j))
                    + alpha * s_old(i, j);
            }
        }
```

Listing 6: OpenMP stencil operator

2.1. Bitwise identical results

Because we use the OpenMP `reduction` clause in `hpc_dot` and `hpc_norm2` we cannot guarantee bitwise identical results. This is because the order of operations is not guaranteed by the standard and can vary between runs. If required be could calculate a tolerance and compare the results within this tolerance. See OpenMP docs for more information.

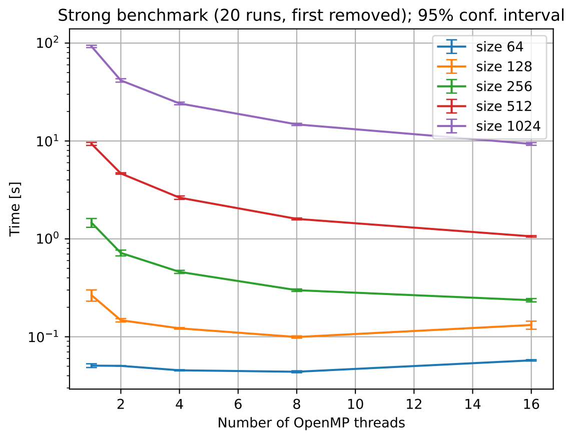
To test this quickly I calculated a quick checksum of the binary data using the `std::hash` function. The code is shown in Listing 7. Runnning the programm with different thread counts and comparing the checksums showed that the results are indeed not bitwise identical.

```
std::size_t hash = std::hash<std::string>{}(
    std::string((char *)f.data(), nx * nx * sizeof(double))
);
```

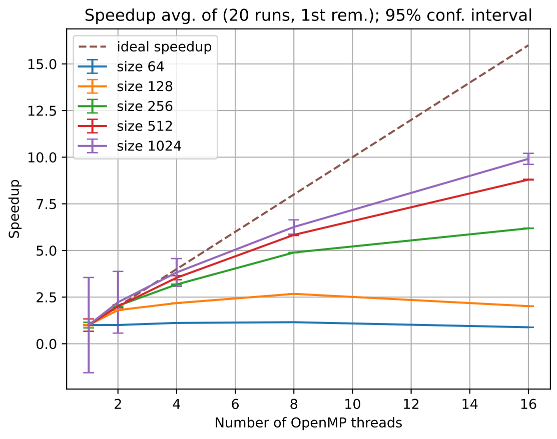
Listing 7: Checksum calculation

2.2. Strong Scaling Benchmark

To evaluate the strong scaling performance of the OpenMP version I ran it on Euler VII — phase 2 (AMD EPYC 7763) using $CPUs = 1, 2, 4, 8, 16$ and increasing image size $N = 64, 128, 256, 512, 1024$. The run configuration I used can be found in the `code` directory. For each configuration I ran 20 runs, excluded the first (warmup) and then took the average of the remaining times. The strong benchmark results are plotted in Figure 2a and the speedup relative to the average serial time is shown in Figure 2b.



(a) Strong scaling benchmark of PDE mini app



(b) Speedup of PDE mini app

Figure 2: Strong scaling and speedup of the PDE mini app

We can observe a decent speedup for sizes 256 and up. As expected for all sizes the curve flattens out as we increase the number of threads. A 10 fold speedup with 16 threads is achieved for $N = 1024$ which is a good result.

The full benchmarking code can be found in the `code` directory and the resulting data in the `code/out` directory.

2.3. Weak Scaling Benchmark

To evaluate the weak scaling performance of the OpenMP version I ran it on Euler VII — phase 2 (AMD EPYC 7763) using $CPUs = 1, 4, 16, 64$ and base image size $N = 64, 128, 256$ increasing it to keep the load per CPU constant. For example, for $CPUs = 4$ and $N = 64$ we would use $N = 128$ for $CPUs = 16$ and so on. The run configuration I used can be found in the `code` directory. For each configuration I ran 20 runs, excluded the first (warmup) and then took the average of the remaining times. The weak benchmark results are plotted in Figure 3 and the efficiency relative to the average serial time is shown in Figure 4.

For the weak benchmark we would ideally see a flat line. This is however rarely the case and indeed, especially in the beginning we observe a relatively strong time increase (notice the log time scale). For higher thread counts however, the increase is not nearly as strong. A similar picture can be seen in the efficiency as we increase the number of CPUs. Unfortunately for all three N and $CPUs = 64$ the variance is quite high and the efficiency results are not reliable. We can however see that we get a sligthly better efficiency for larger N. For $CPUs = 16$ we still achieve around 14% efficiency with the largest base size.

Again the full benchmarking code can be found in the `code` directory and the resulting data in the `code/out` directory.

All source code can be found in `code/`.

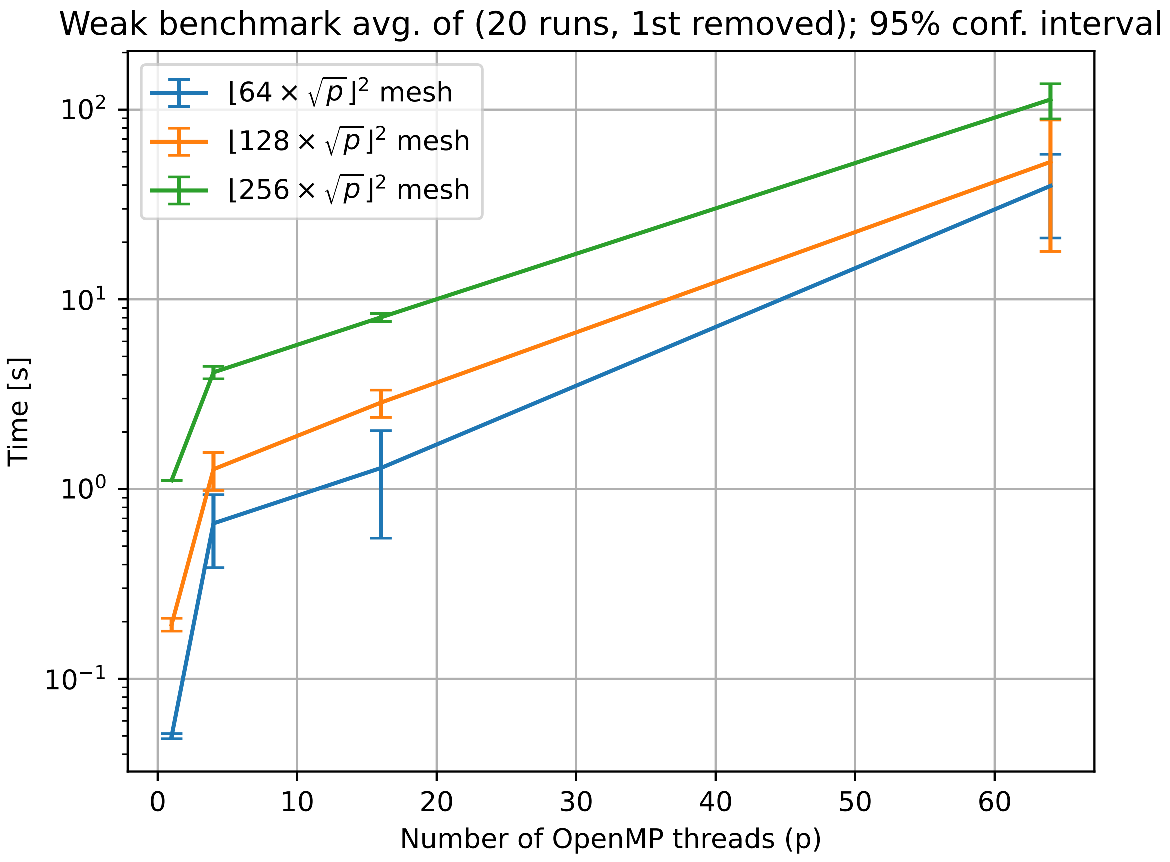


Figure 3: Weak scaling benchmark of PDE mini app

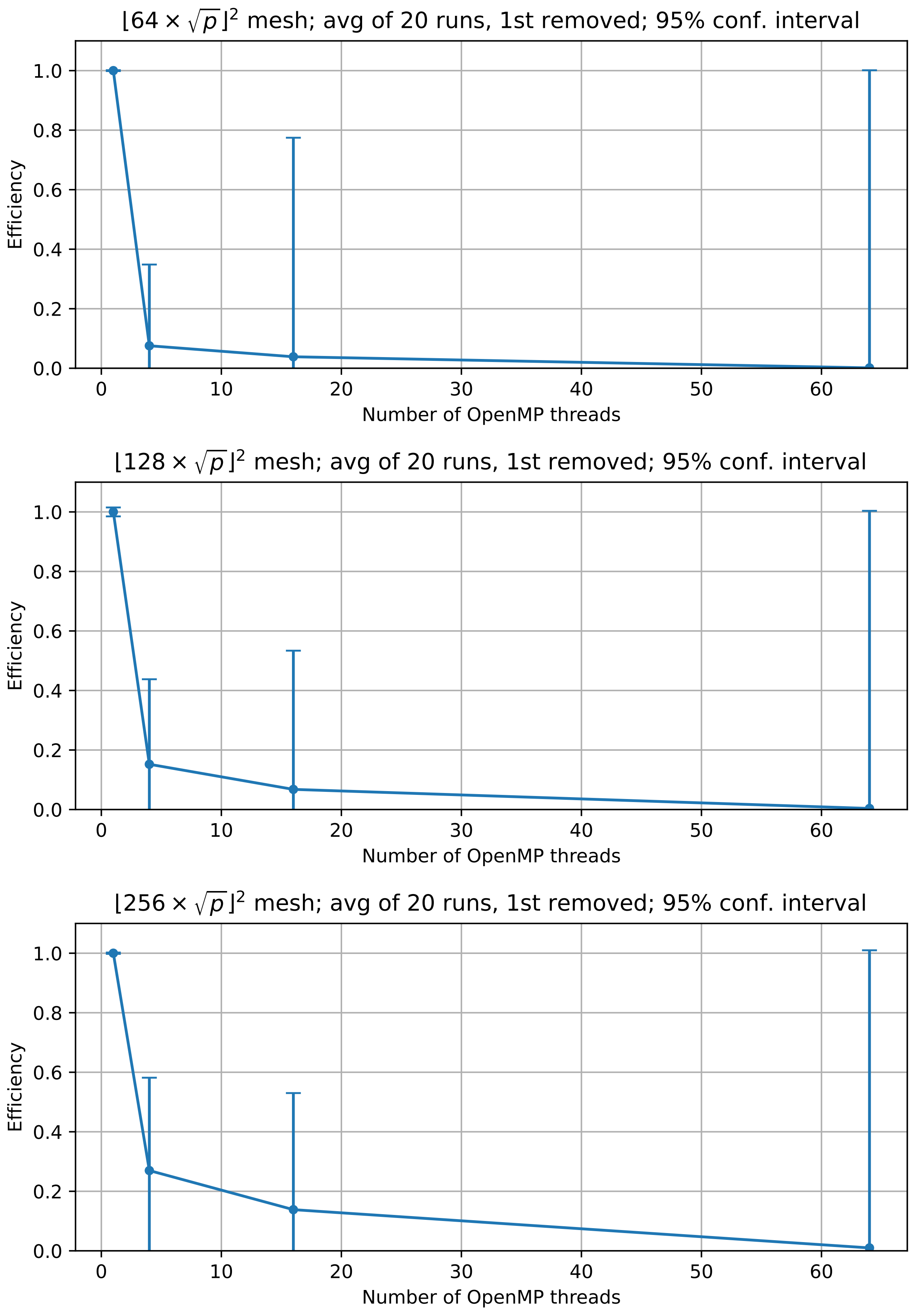


Figure 4: Efficiency of PDE mini app