Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Qianbo Zang Discussed with: FULL NAME

Solution for Project 3

Due date: 02.11.2022, 23:59

HPC 2022 — 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. Open the file linalg.cpp and implement the functions hpc xxxxx(). Follow the comments in the code as they are there to help you with the implementation. [18 Points]

Finish $hpc\_norm2()$: $x = \sqrt{< x, x >}$

```
result = sqrt(hpc_dot(x, x, N));
```

Finish $hpc\_fill()$:

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

Finish $hpc\_axpy()$:

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

Finish *hpc_add_scaled_diff*():

```
for (int i = 0; i < N; ++i)
{
    y[i] = x[i] + alpha * (l[i] - r[i]);
}
```

Finish *hpc_scaled_diff*():

```
for (int i = 0; i < N; ++i)
{
    y[i] = alpha * (l[i] - r[i]);
}
```

Finish *hpc_scale*():

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

Finish *hpc_lcomb*()

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

Finish *hpc_copy*()

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

1.2. Open file operators.cpp and implement the missing stencil kernels. [15 Points]

The interior grid points

$$f_{i,j} = [-(4+\alpha)s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j}(1 - s_{i,j})]^{k+1} + \alpha s_{i,j}^{k} = 0$$

```
f(i, j) = - (4.0 + alpha) * s(i, j)
          + s(i-1, j) + s(i+1, j)
          + s(i, j-1) + s(i, j+1)
          + beta * s(i, j) * (1.0 - s(i, j))
          + alpha * y_old(i, j);
```

The west boundary: the point of west boundary to (i, j) is (i-1, j). So, modify s(i-1, j) to bndW[j].

```
f(i, j) = - (4.0 + alpha) * s(i, j)
          + bndW[j] + s(i+1, j)
          + s(i, j-1) + s(i, j+1)
          + beta * s(i, j) * (1.0 - s(i, j))
          + alpha * y_old(i, j);
```

The inner north boundary: the point of inner north boundary is in the source side of outside boundary, which is (i, j+1).

```
for (int i = 1; i < iend; ++i)
{
    f(i, j) = - (4.0 + alpha) * s(i, j)
              + s(i-1, j) + s(i+1, j)
              + s(i, j-1) + bndN[i]
```

```
                + alpha * y_old(i,j)
                + beta * s(i,j) * (1.0 - s(i,j));
}
```

The inner south boundary:

```
for (int i = 1; i < iend; i++)
{
    f(i,j) = -(4.0 + alpha) * s(i, j)
                + s(i-1, j) + s(i+1, j)
                + bndS[i] + s(i, j+1)
                + alpha * y_old(i, j)
                + beta * s(i, j) * (1.0 - s(i, j));
}
```

1.3. Compare the number of conjugate gradient iterations and the Newton iterations with the reference output above.

```
[stud46@icsnode30 serial]$ ./main 128 100, 0.005
================================================================================
                        Welcome to mini-stencil!
version   :: Serial C++
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.223947 seconds
1511 conjugate gradient iterations, at rate of 6747.12 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Figure 1: The terminal of serial

1.4. Plot the solution with the script plotting.py and include it in your report. It should look like in Figure 3. [2 Points]

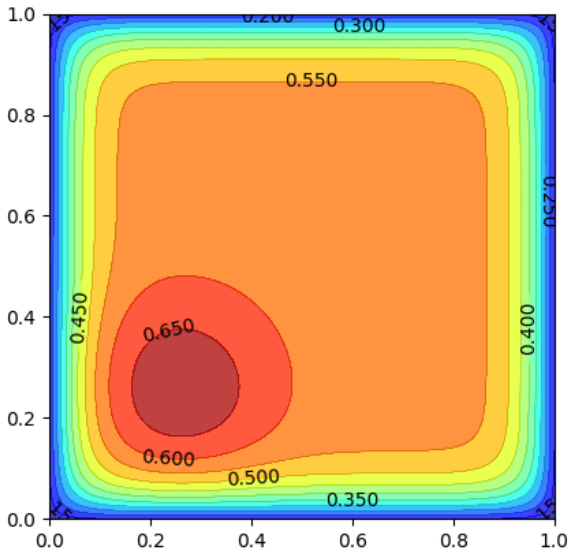

Figure 2: Output of the mini-app for a domain discretization into 128x128 grid points, 100 time steps with a simulation time from 0-0.005 s

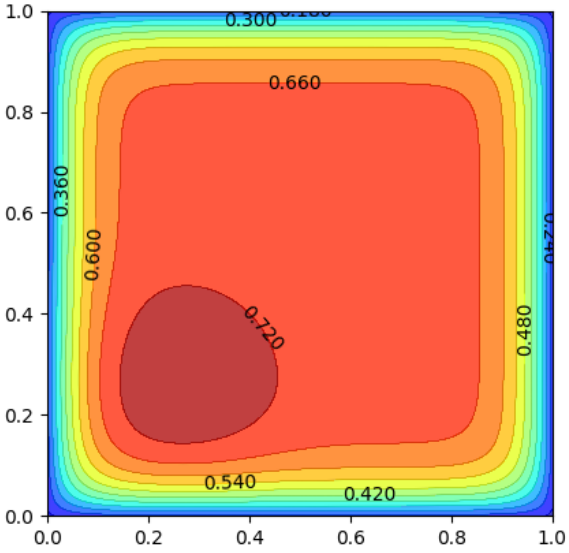

Figure 3: Output of the mini-app for a domain discretization into 128x128 grid points, 100 time steps with a simulation time from 0-0.006 s

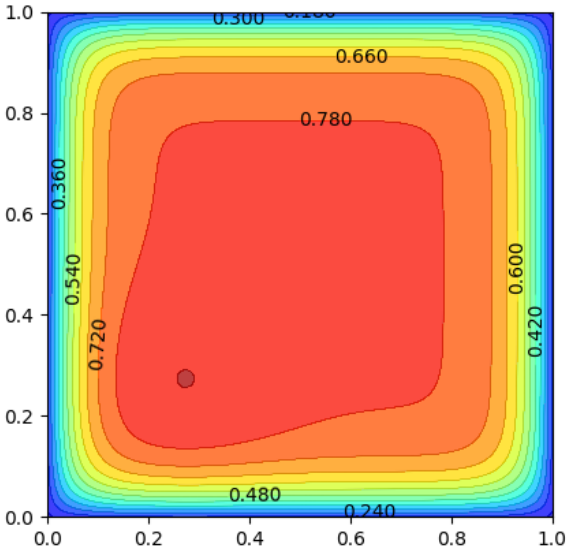

Figure 4: Output of the mini-app for a domain discretization into 128x128 grid points, 100 time steps with a simulation time from 0-0.007 s

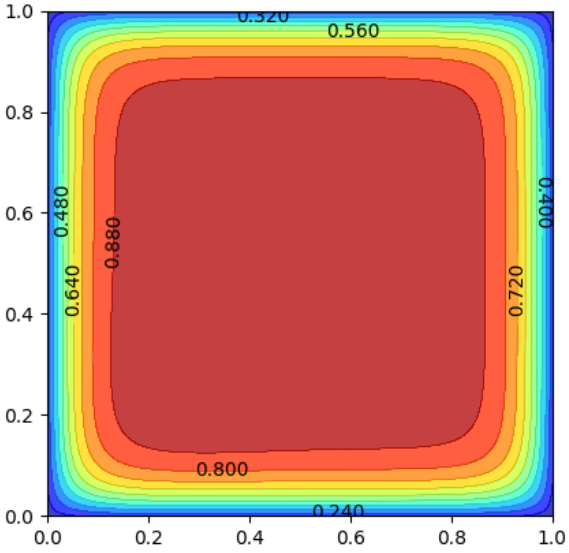

Figure 5: Output of the mini-app for a domain discretization into 128x128 grid points, 100 time steps with a simulation time from 0-0.008 s

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

2.1. Linear algebra kernel [15 Points]

```
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;
}

// computes the 2-norm of x
// x is a vector on length N
double hpc_norm2(Field const& x, const int N)
{
    double result = 0;

    //TODO
    result = hpc_dot(x, x, N);

    return sqrt(result);
}

// sets entries in a vector to value
// x is a vector on length N
// value is a scalar
void hpc_fill(Field& x, const double value, const int N)
{
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        x[i] = value;
    }
```

```
}

//////////////////////////////////////////////////////////////////////////////
//  blas level 1 vector-vector operations
//////////////////////////////////////////////////////////////////////////////

// computes y := alpha*x + y
// x and y are vectors on length N
// alpha is a scalar
void hpc_axpy(Field& y, const double alpha, Field const& x, const int N)
{
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        y[i] = alpha * x[i] + y[i];
    }
}

// computes y = x + alpha*(l-r)
// y, x, l and r are vectors of length N
// alpha is a scalar
void hpc_add_scaled_diff(Field& y, Field const& x, const double alpha,
    Field const& l, Field const& r, const int N)
{
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        y[i] = x[i] + alpha * (l[i] - r[i]);
    }
}

// computes y = alpha*(l-r)
// y, l and r are vectors of length N
// alpha is a scalar
void hpc_scaled_diff(Field& y, const double alpha,
    Field const& l, Field const& r, const int N)
{
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        y[i] = alpha * (l[i] - r[i]);
    }
}

// computes y := alpha*x
// alpha is scalar
// y and x are vectors on length n
void hpc_scale(Field& y, const double alpha, Field const& x, const int N)
{
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        y[i] = alpha * x[i];
    }
}

// computes linear combination of two vectors y := alpha*x + beta*z
// alpha and beta are scalar
// y, x and z are vectors on length n
void hpc_lcomb(Field& y, const double alpha, Field const& x, const double beta,
    Field const& z, const int N)
{
```

```
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        y[i] = alpha * x[i] + beta * z[i];
    }
}

// copy one vector into another y := x
// x and y are vectors of length N
void hpc_copy(Field& y, Field const& x, const int N)
{
    //TODO
    #pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        y[i] = x[i];
    }
}
```

2.2. The diffusion stencil [10 Points]

```
#pragma omp parallel shared(f, s, y_old, bndE, bndN, bndS, bndW)
{
    // the interior grid points
    #pragma omp for collapse(2)
    for (int j=1; j < jend; j++) {
        for (int i=1; i < iend; i++) {
            //TODO
            // f(i,j) = ...
            f(i, j) = - (4.0 + alpha) * s(i, j)
                      + s(i-1, j) + s(i+1, j)
                      + s(i, j-1) + s(i, j+1)
                      + beta * s(i, j) * (1.0 - s(i, j))
                      + alpha * y_old(i, j);
        }
    }

    // the east boundary
    {
        int i = nx - 1;
        #pragma omp for
        for (int j = 1; j < jend; j++)
        {
            f(i,j) = -(4. + alpha) * s(i,j)
                      + s(i-1,j) + s(i,j-1) + s(i,j+1)
                      + alpha*y_old(i,j) + bndE[j]
                      + beta * s(i,j) * (1.0 - s(i,j));
        }
    }

    // the west boundary
    {
        int i = 0;
        //TODO
        #pragma omp for
        for (int j = 1; j < jend; ++j)
        {
            f(i, j) = - (4.0 + alpha) * s(i, j)
                      + bndW[j] + s(i+1, j)
                      + s(i, j-1) + s(i, j+1)
                      + beta * s(i, j) * (1.0 - s(i, j))
                      + alpha * y_old(i, j);
        }
    }
```

```
// the north boundary (plus NE and NW corners)
{
    int j = nx - 1;

    {
        int i = 0; // NW corner
        f(i,j) = -(4. + alpha) * s(i,j)
                    + s(i+1,j) + s(i,j-1)
                    + alpha * y_old(i,j) + bndW[j] + bndN[i]
                    + beta * s(i,j) * (1.0 - s(i,j));
    }

    // inner north boundary
    //TODO
    #pragma omp for
    for (int i = 1; i < iend; ++i)
    {
        f(i, j) = - (4.0 + alpha) * s(i, j)
                    + s(i-1, j) + s(i+1, j)
                    + s(i, j-1) + bndN[i]
                    + alpha * y_old(i,j)
                    + beta * s(i,j) * (1.0 - s(i,j));
    }

    {
        int i = nx-1; // NE corner
        f(i,j) = -(4. + alpha) * s(i,j)
                    + s(i-1,j) + s(i,j-1)
                    + alpha * y_old(i,j) + bndE[j] + bndN[i]
                    + beta * s(i,j) * (1.0 - s(i,j));
    }
}

// the south boundary
{
    int j = 0;

    {
        int i = 0; // SW corner
        f(i,j) = -(4. + alpha) * s(i,j)
                    + s(i+1,j) + s(i,j+1)
                    + alpha * y_old(i,j) + bndW[j] + bndS[i]
                    + beta * s(i,j) * (1.0 - s(i,j));
    }

    // inner south boundary
    //TODO
    #pragma omp for
    for (int i = 1; i < iend; i++)
    {
        f(i,j) = - (4.0 + alpha) * s(i, j)
                    + s(i-1, j) + s(i+1, j)
                    + bndS[i] + s(i, j+1)
                    + alpha * y_old(i, j)
                    + beta * s(i, j) * (1.0 - s(i, j));
    }

    {
        int i = nx - 1; // SE corner
        f(i,j) = -(4. + alpha) * s(i,j)
                    + s(i-1,j) + s(i,j+1)
                    + alpha * y_old(i,j) + bndE[j] + bndS[i]
                    + beta * s(i,j) * (1.0 - s(i,j));
    }
}
```

```
}
```

```
[stud46@icsnode27 parallel]$ ./main 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 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.0772394 seconds
1511 conjugate gradient iterations, at rate of 19562.5 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Figure 6: The terminal of parallel, ./main 128 100 0.005

```
[stud46@icsnode27 parallel]$ ./main 128 100 0.01
================================================================================
                        Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 8
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.01
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.105782 seconds
2229 conjugate gradient iterations, at rate of 21071.6 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Figure 7: The terminal of parallel, ./main 128 100 0.01

2.3. Strong scaling [10+3 Points]

2.3.1. Plot

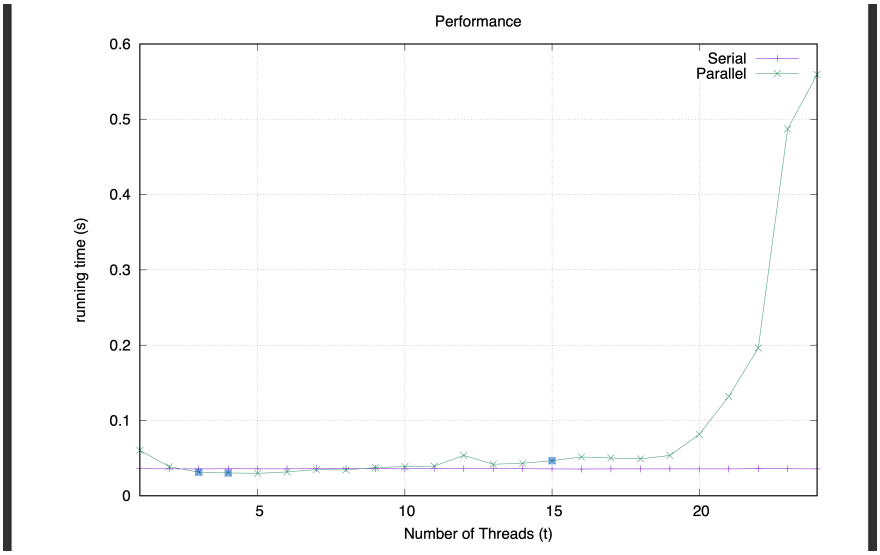

Figure 8: 64 x 64

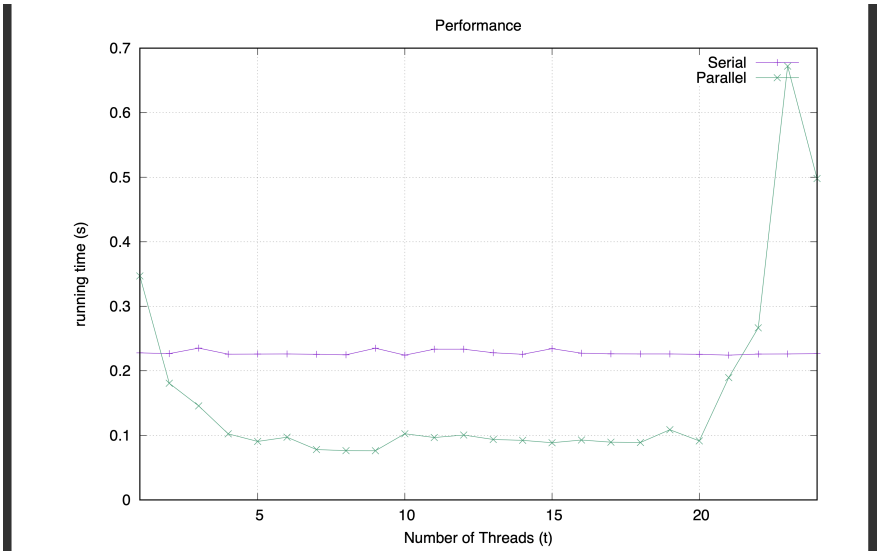

Figure 9: 128 x 128

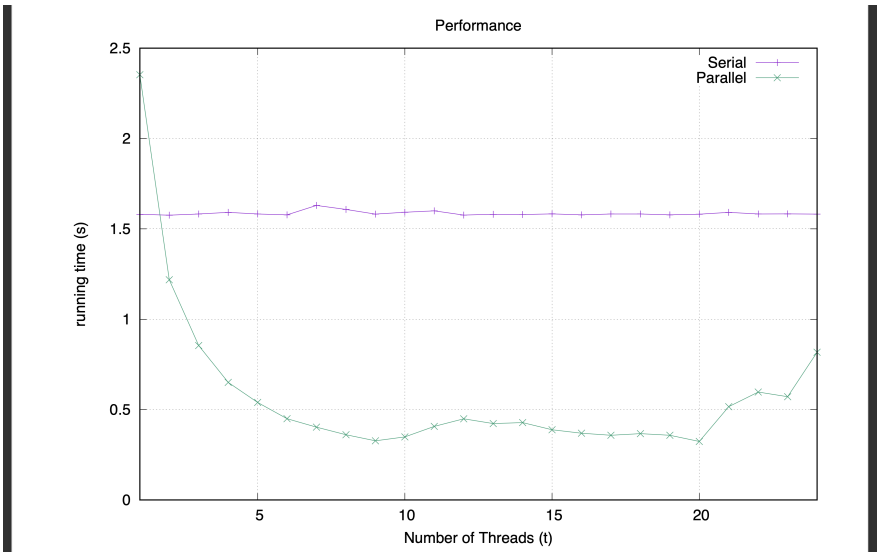

Figure 10: 256 x 256

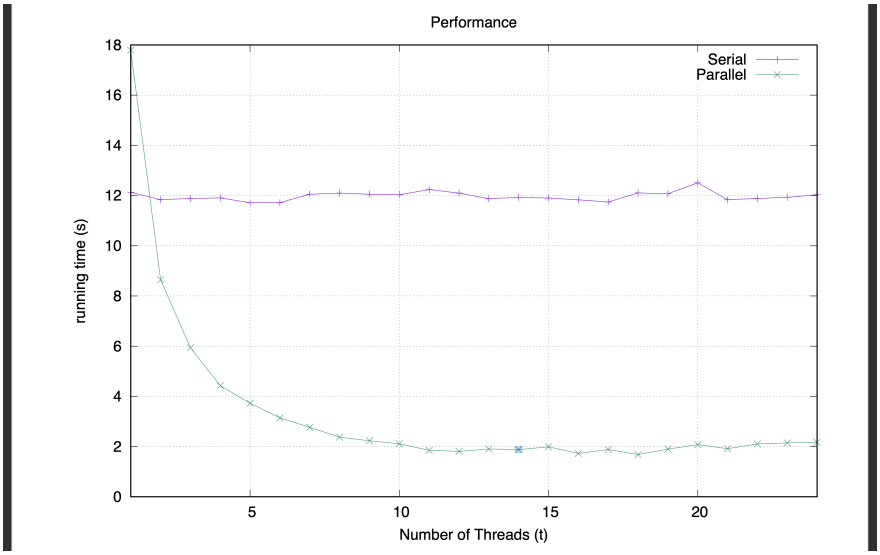

Figure 11: 512 x 512

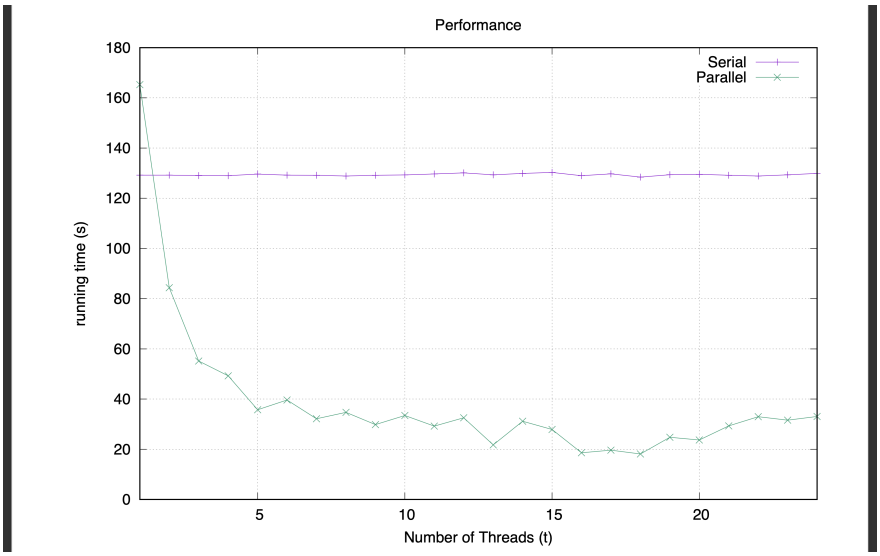

Figure 12: 1024 x 1024

2.3.2. Analysis

If the number of meshes is small (64 and 128), the parallelisation will not increase performance. In fact, too many threads only decrease the performance. In parallelisation, data transfer between different threads requires communication time. When the amount of data is small, the communication time may higher than computation time, resulting in a slower performance after the parallelisation. If the number of meshes is large (512 and 1024), the parallelisation will save $\frac{4}{5}$ time in this case.

Additionally, "Can a threaded OpenMP PDE solver be implemented which produces bitwise-identical results without any parallel side effects or not". The answer is False. Because there are a lot of floating point operations in operations. Floating-point operations are inexact. In the parallelisation, the changes in the order of operations (such as +, -) cause uncertainty in the result.

2.4. Weak scaling [10 Points]

When "export OMP_NUM_THREADS=4" and "./main 64 100 0.005", the simulation time is 0.0297996. When "export OMP_NUM_THREADS=16" and "./main 128 100 0.005", the simu-

lation time is 0.787382. When "export OMP_NUM_THREADS=64" and "./main 256 100 0.005", the simulation time is 4.49279. Theoretically, these three should have same simulation time but the time of 64-mesh case is smallest.

The first reason is the increase of conjugate gradient iterations from 1007 to 1514 to 2786. When the number of meshes doubles, it doesn't just mean that the amount of computation becomes quadruple. The number of optimization loops also increases.
The second reason is the increase of communication time. The data transfer between different threads requires communication time. And when the number of threads increases, the communication time is not simply multiplied.

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

4. Bonus Question [5-10 Points] Can you make your implementation even faster by using SIMD instructions?

In this case, SIMD will increase performance, when the numer of meshes is 256 and 512. But the effect is not obvious. Because although the addition of two vectors to form a third vector is a typical SIMD operation, vector multiplication is not a SIMD operation. Not all operations can be accelerated by SIMD.

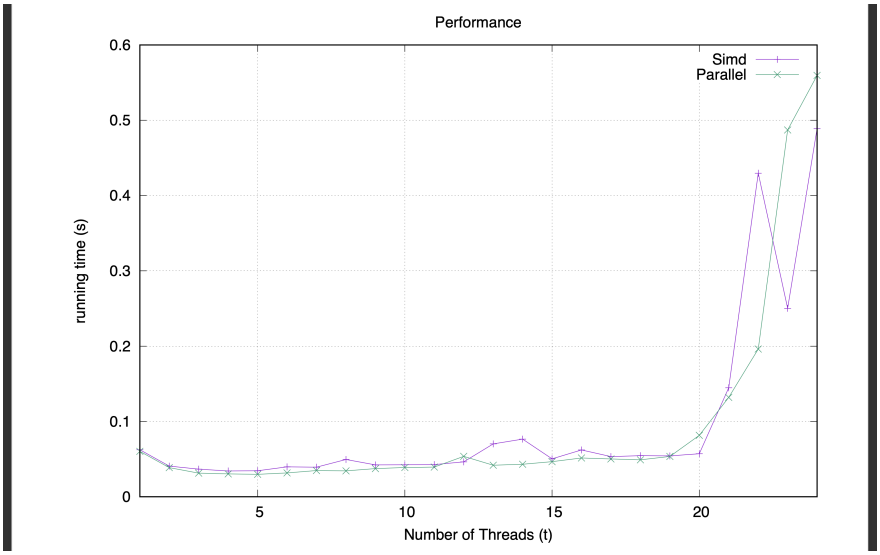

Figure 13: 64 x 64

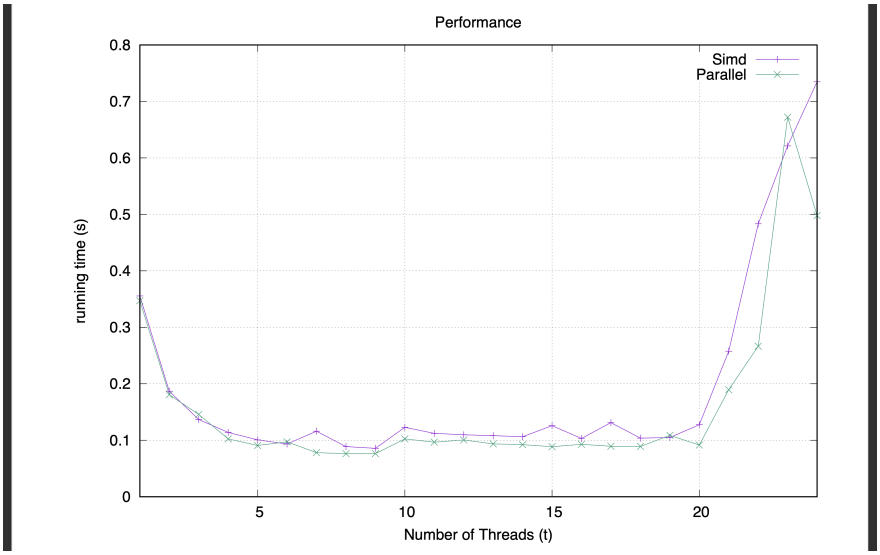

Figure 14: 128 x 128

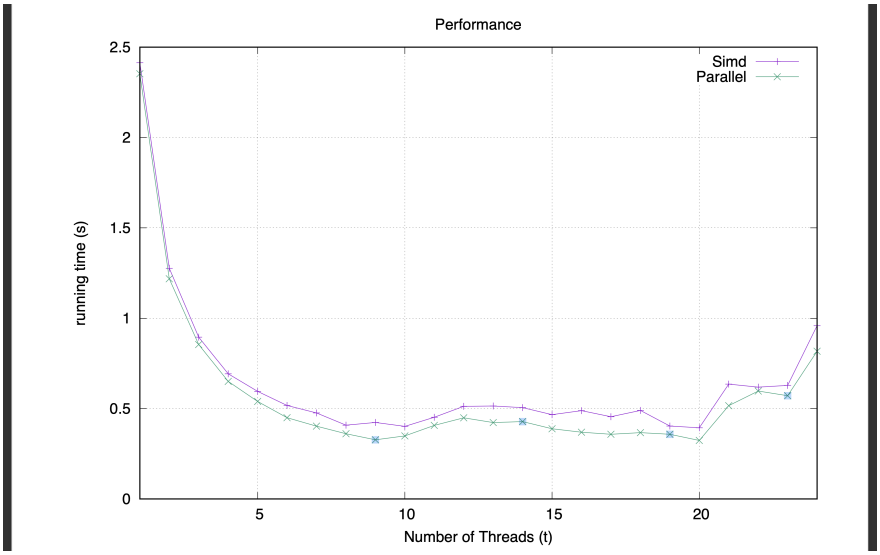

Figure 15: 256 x 256

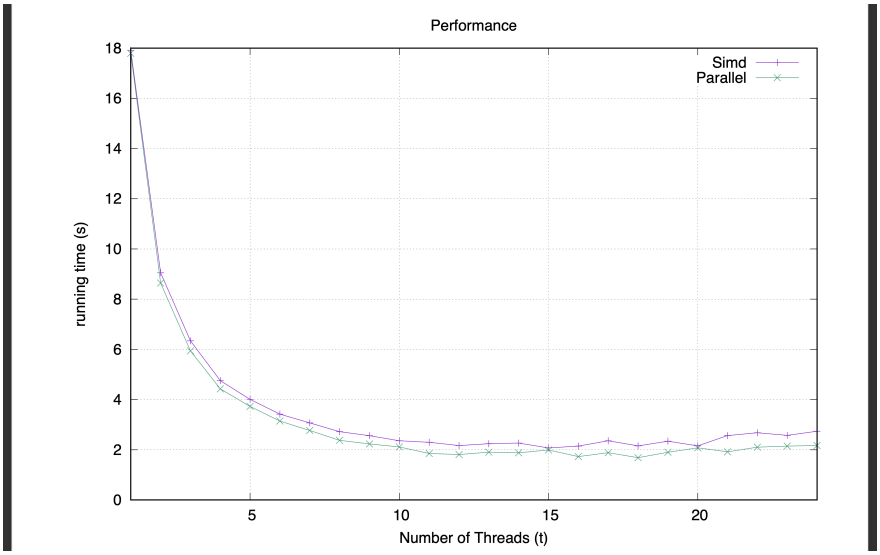

Figure 16: 521 x 512

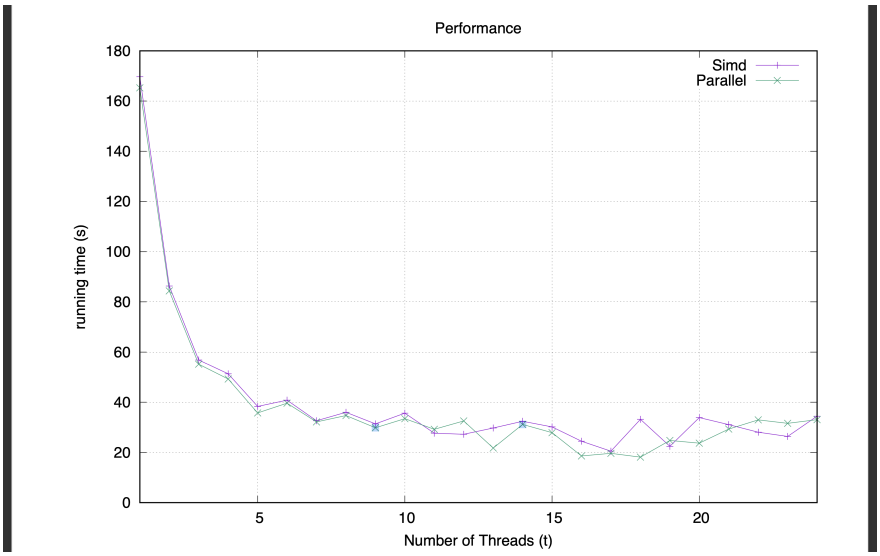

Figure 17: 1024 x 1024

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.