Università della Svizzera italiana

Institute of Computational Science ICS

High-Performance Computing Lab **2020**

Student: Gabriele Berra Discussed with: -

Solution for Project 4

Due date: 30.10.2020 (midnight)

HPC Lab 2020 — 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.

In this fourth assignment I have to complete a code in order to solve the Fisher's equation and, after that, parallelize it using openMP. The Fisher equation,

$$\frac{\partial s}{\partial t} = D\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right) + Rs(1-s)$$

was first proposed by Ronald Fisher in 1937 in order to describe the dynamics of advantageous genes.
In our specific case we discretise its domain using a uniform grid in which we approximate the points with a second order finite difference approximation of the type:

$$\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right)_{i,j} \approx \frac{1}{\Delta x^2}(-4s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1})$$

And, for the time derivative:

$$\left(\frac{\partial s}{\partial t}\right)^k_{i,j} \approx \frac{1}{\Delta t}(s^k_{i,j} - s^{k-1}_{i,j})$$

In order to obtain an approximation we can reformulate the two equations above as follows:

$$f^k_{i,j} := [-(4+\alpha)s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j}(1-s_{i,j})]^k + \alpha s^{k-1}_{i,j} = 0 \qquad (1)$$

. Since each equation is quadratic in s we have to solve it using Newton's method for the roots of a function. The Newton's method is based on iteratively search the root of a function taking in considerations the derivatives and an initial guess x_0. Since we have n^2 derivatives for a given k, with the aim of calculating the inverse Jacobian matrix, we use a Conjugate Gradient solver to solve the linear system of equation given by rearranging the terms of the Newton's method:

$$[\mathbf{J_f}(\mathbf{y}^l)]\delta\mathbf{y}^{l+1} = \mathbf{f}(\mathbf{y}^l)$$

1. Task: Implementing the linear algebra functions and the stencil operators [40 Points]

For this first task of the assignment 4 I have, on one hand, to implement the functions in the *linalg.cpp* file and, on the other, to implement the missing stencil in the *operators.cpp* file. I started tackling the implementation of the linear algebra functions by following the instructions given in the code. The script I implemented is the following:

```
////////////////////////////////////////////////////////////////////////
//  blas level 1 reductions
////////////////////////////////////////////////////////////////////////

// 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;
    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;
    for (int i = 0; i < N; i++) {
        result += x[i] * x[i];
    }
    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) {
    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
    ) {
    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) {
    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) {
    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& x, const int N) {
    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& x, const double
    beta,
               Field const& z, const int N) {
    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) {
    for (int i = 0; i < N; i++) {
        y[i] = x[i];
    }
}
```

In the *operators.cpp* file I have to implement: the nested for loop for the interior grid points, the west boundary, inner north boundary and, inner south boundary. To do so I firstly have to understand the difference between the positions of the points in the grid and how to better calculate their approximation. We know from the theory that a generic inner point in the grid can be easily calculated with equation 1. For the boundaries, I have to modify equation 1 since, on these points, some neighbours are missing (e.g. on the south-east corner the "south" and the "east" point are missing). The modified equations are the following:

For the west boundary:

```
    int i = 0;
    for (int j = 1; j < jend; j++) {
      f(i, j) = -(4. + 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);
    }
```

For the inner north boundary:

```
    for (int i = 1; i < iend; i++) {
      f(i, j) = -(4. + alpha) * s(i, j)
                + s(i - 1, j) + s(i + 1, j)
                + s(i, j - 1) + bndN[i]
                + beta * s(i, j) * (1.0 - s(i, j))
                + alpha * y_old(i, j);

    }
```

For the inner south boundary:

```
    for (int i = 1; i < iend; i++) {
      f(i, j) = -(4 + alpha) * s(i, j)
                + s(i - 1, j) + s(i + 1, j)
                + bndS[i] + s(i, j + 1)
                + beta * s(i, j) * (1.0 - s(i, j))
                + alpha * y_old(i, j);
    }
```

2. Task: Adding OpenMP to the nonlinear PDE mini-app [60 Points]

This second part of the assignment is divided into 5 subsections mainly focused on the parallelization of the code of the first part. These subsections are:

1. Replace welcome message in main.cpp.
2. Linear algebra kernel.
3. The diffusion stencil.

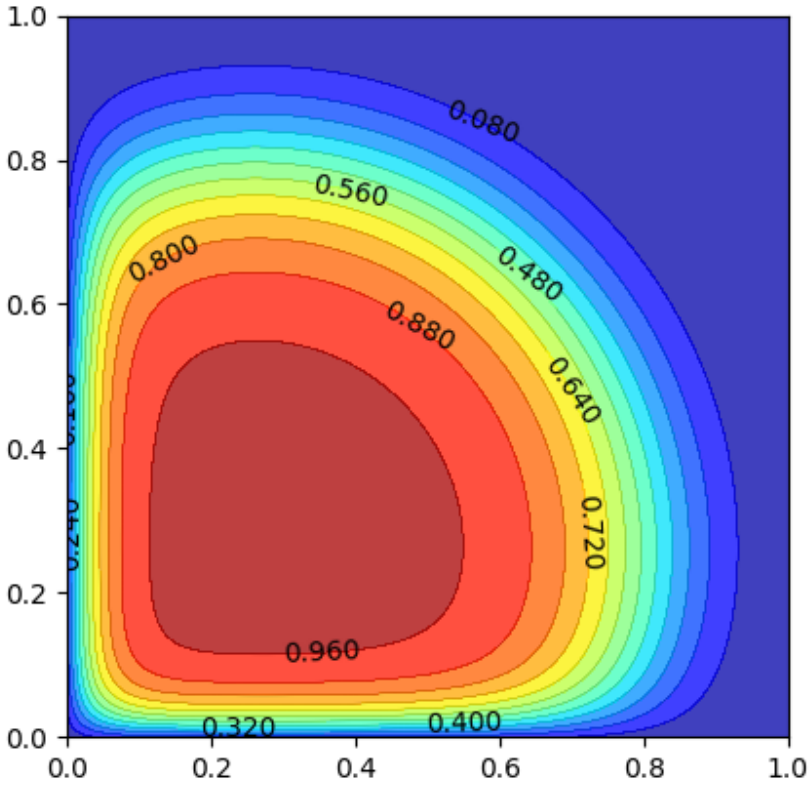

Figure 1: Serial plot with 128 x 128 grid points

4. Strong scaling.

5. Weak scaling.

For the first point, the main issue - in my opinion - is how to "connect" the environment variable *export OMP_NUM_THREADS=x* with a local variable in the *main.cpp* script. I decided to solve this problem by adding the command *omp_get_max_threads()*, which gives as output the maximum number of threads that can be used in the parallel regions. I implemented it in the script as follows:

```
std::cout << "
   ================================================================================"
   << std::endl;
std::cout << "                              Welcome to mini-stencil!" <<
   std::endl;
std::cout << "version    :: C++ OpenMP" << std::endl;
std::cout << "threads    :: " << omp_get_max_threads() << std::endl;
std::cout << "mesh       :: " << options.nx << " * " << options.nx
   << " dx = " << options.dx << std::endl;
std::cout << "time       :: " << nt << " time steps from 0 .. " <<
   options.nt*options.dt << std::endl;;
std::cout << "iteration  :: " << "CG "           << max_cg_iters
          << ", Newton "     << max_newton_iters
          << ", tolerance " << tolerance << std::endl;;
```

```
    std::cout << "
       ==========================================================================="
       << std::endl;
```

For the second point I have to implement parallel section only in the linear algebra script. Since all the functions that I added have a *for loop* inside, I decided to use the *# pragma omp parallel for* command in all of the functions. By doing so, I noticed that in *hpc_dot()* and *hpc_norm2()* adding only a *parallel for* section is not enough to perform the correct calculations. Thus, since the variable *result* is just a sum over the iterations of the previous loop, I used a *reduction* clause to perform the parallelization. The code is reported below.

```
////////////////////////////////////////////////////////////////////
//  blas level 1 reductions
////////////////////////////////////////////////////////////////////

// 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;
}

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

    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) {
    #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
    ) {
    #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) {
    #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) {
    #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& x, const int N) {
    #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& x, const double
    beta,
               Field const& z, const int N) {
    #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) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        y[i] = x[i];
    }
}
```

After the parallelization of the *linalg_omp.cpp* script, I expect an increase in the performance since most of the operations are performed using linear algebra functions. Comparing the results of this first step of parallelization - using 8 threads - with the serial version we can notice an improvement, in terms of performance - i.e. speed - both for the 128 x 128 grid and the 256 x 256 one. The results are reported in Table 1 (time expressed in seconds):

Grid Size	**Serial Time**	**Parallel Time**
128 x 128	1.22729	0.760295
256 x 256	9.01825	3.72461

Table 1: Time with the serial version of *linalg.cpp* and the parallel one using 8 threads

The next step is to parallelize the *operations_omp.cpp* script. The parallelization of this script is non-trivial since we have to understand which of the for loops has to be parallel and which one does not. In my opinion, we have to think about how many iterations each loop has to do in order to achieve the correct result. If we consider a 10 x 10 grid, the inner point for which the script has to do the calculations are 64 while, in contrast, the boundary points are "only" 38. If we take in consideration a bigger grid, for example a 128 x 128, the points on the boundary are 508 and the inner points are 15'876. So for a bigger grid the nested loop - i.e. the loop that performs the operation in the inner point - has a major impact on the general performance and, thus, I focused my attention on the parallelization of the latter one. The parallelized section of the code is reported below.

```
    #pragma omp parallel for
    for (int j = 1; j < jend; j++) {
        for (int i = 1; i < iend; i++) {
            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) * (1.0 - s(i, j))
                      + alpha * y_old(i, j);

        }
    }
```

2.1. Strong scaling

As we know from the theory the *strong scaling* is defined as the scalability of a specific threaded algorithm when the number of threads used increases and the problem size remains constant. In our case I tested the algorithm for different sizes and, for each, I tested the strong scalability. The results are summarized in the following Table 2 and Fig. 2 .

In general, as we can see in Fig. 2, for any given size the code scales except for the 64 x 64 grid in which, an increase from 9 to 10 threads produces a worst performance compared to the

# threads/Grid Size	64 x 64	128 x 128	256 x 256	512 x 512	1024 x 1024
1	0.196281	1.28041	8.81197	65.0061	717.519
2	0.187238	0.953788	6.15223	45.2533	408.307
3	0.162066	0.835927	5.02967	36.4704	345.568
4	0.165711	0.77066	4.43689	30.5576	346.456
5	0.175701	0.756458	4.1095	28.737	357.694
6	0.171196	0.752386	3.9769	26.9436	320.769
7	0.184268	0.768854	3.78487	26.2863	298.011
8	0.192468	0.773505	3.86266	24.5194	303.903
9	0.199429	0.763555	3.70059	25.2214	288.606
10	0.268303	0.776758	3.61467	23.8828	301.479

Table 2: Strong scaling

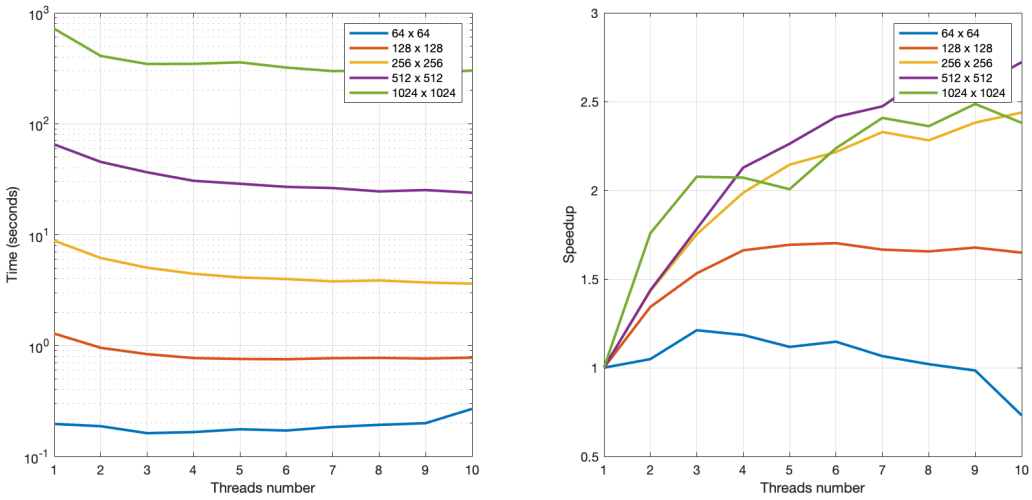

Figure 2: Strong scaling and speedup

one using only one thread. This is probably because the time for dividing the tasks between the threads in the parallel section and the computation time is greater than the computation time with one thread. In all the other cases is beneficial the usage of more threads, in particular for big grid sizes.

2.2. Weak scaling

For this last section I will tackle the analysis of the weak scalability, which can be defined as follows: how the solution time varies with the number of processors for a fixed problem size per processor.[1] In my analysis I decided to use 1, 4 and 16 threads with problem sizes associated to each thread given by:

- 64 x 64 for each thread, with the total problem size going from 64 x 64 (1 thread) to 128 x 128 (4 threads) and 256 x 256 (16 threads)
- 128 x 128 for each thread, with the total problem size going from 128 x 128 (1 thread) to 256 x 256 (4 threads) and 512 x 512 (16 threads)
- 256 x 256 for each thread, with the total problem size going from 256 x 256 (1 thread) to 512 x 512 (4 threads) and 1024 x 1024 (16 threads)

[1]Source: https://en.wikipedia.org/wiki/Scalability#Weak_versus_strong_scaling

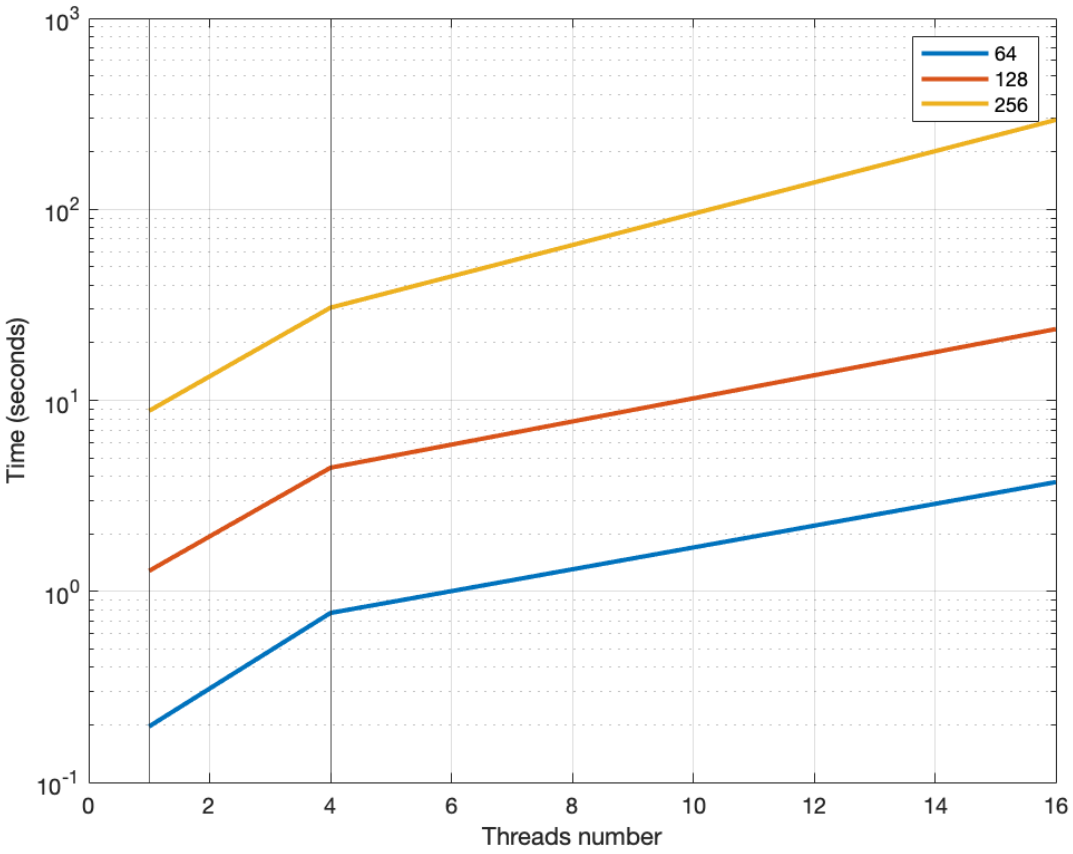

Figure 3: Weak scaling with different initial size

As we can notice in Fig.3 the time for the computation slightly increases due to the overhead of the threads since, for a constant problem size assigned to each thread, the time for the distribution of the tasks inside the parallel region increases, this cause the plot to be non-constant.