Contents

I. Theory

1. The Diffusion Equation

We consider the process of diffusion of e.g. heat in a solid material or solutes in a solvent. Diffusion can be modeled by a second order, linear partial differential equation, and we will investigate several ways to solve this equation numerically and simulate it on parallel computers. Despite the fact that the model itself is not too complicated, solving it numerically and simulating it on a parallel computers forces us to touch upon many important themes of parallel scientific computing that are also encountered in more complicated mathematical models
The diffusion equation is given by:

$$\frac{\partial c}{\partial t} = \mathbf{D}\nabla^2 c \tag{1}$$

Let us now consider the two-dimensional case, i.e.

$$\frac{\partial c}{\partial t} = \mathbf{D}(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}) \tag{2}$$

We will not come into detail the derivation, the discretisation constraint is given by:

$$\frac{\partial^2 c^n_{l,m}}{\partial x^2} = \frac{c^n_{l-1,m} - 2c^n_{l,m} + c^n_{l+1,m}}{\delta x^2} \tag{3}$$

and similar for the derivatives in y :

$$\frac{\partial^2 c^n_{l,m}}{\partial y^2} = \frac{c^n_{l,m-1} - 2c^n_{l,m} + c^n_{l,m+1}}{\delta x^2} \tag{4}$$

2. Time Independent Diffusion Equation

In many cases one is only interested in the final steady state concentration field and not so much in the transient behavior, i.e. the route towards the steady state is not relevant. This may be so because the diffusion is a very fast process in comparison with other processes in the system, and then one may neglect the transient behavior.
Setting all time derivatives to zero in the diffusion equation (1) we find the time independent diffusion equation:

$$\nabla^2 c = 0 \tag{5}$$

Consider the two-dimensional domain and the discretisation case. It is easy to derive that the solution to discretisation case can be reformulated as

$$\mathbf{Ax=b} \tag{6}$$

To solve this equation we can use either the direct method or iterative method. But the later is cheaper and typically easier to parallelise. We will use the Successive Over Relaxation (SOR) which is one type of iterative method. The fomula for the SOR method is:

$$c^{n+1}_{l,m} = (1-\omega)c^n_{l,m} + \frac{\omega}{4}[(c^n_{l+1,m} + c^{n+1}_{l-1,m} + c^n_{l,m+1} + c^{n+1}_{l,m-1})] \tag{7}$$

The parameter ω determines the strengh of the mixing. It turns out that for finite difference schemes SOR is very advantageous and gives much faster convergence then other iterative method like Jacobi and Gauss-Seidel.

3. Diffusion Limited Aggregation

Diffusion Limited Aggregation (DLA) is a model for non-equilibrium growth, where growth is determined by diffusing particles. It can model e.g. a Bacillus subtilis bacteria colony in a petri dish. The idea is that the colony feeds on nutrients in the immediate environment, that the probability of growth is determined by the concentration of nutrients and finally that the concentration of nutrients in its turn is determined by diffusion. The basic algorithm is:

1. Solve Laplace equation to get distribution of nutrients, assume that the object is a sink (i.e. c = 0 on the object)
2. Let the object grow
3. Go back to (1)

The first step in above algorithm is done by a parallel SOR iteration.

II. Implementation

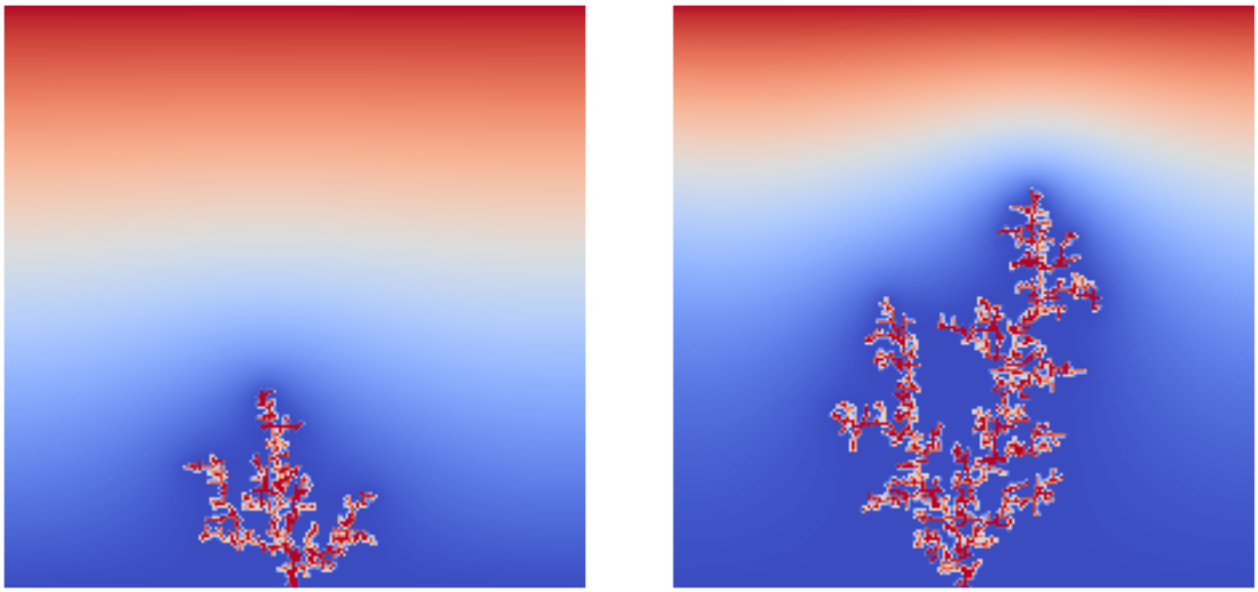
1. Simulation

In this project we simulate a Bacillus subtilis bacteria colony in a petri dish. We define some conditions:

1. Periodic boundary conditions in x-direction
2. Fixed value for the upper and lower boundary in y-direction
3. The object start at the bottom center of the grid

For the hyperparameters we use the following values:

Number of iteration :	800 & 2000
Size of grid :	200x 200
Tolerance value for convergence :	0.001
Omega :	1.9
Upper boundary :	c0 = 1
Lower boundary :	cN = 0



(a) 800 Iterations (b) 2000 Iterations

Figure 1: Results of Serial DLA growth on a 200^2 lattice.

2. Serial Code

We begin with serial version of the code.Let us go into details some main functions. The results is in Figure 1

2..1 Initialization

```
void KhoiTao(float *C_old, float *C_current, int *O, int *candidates)
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            if (i == 0)
                *(C_old + i * N + j) = 1;
            else
                *(C_old + i * N + j) = 0;
            *(C_current + i * N + j) = *(C_old + i * N + j);
            *(O + i * N + j) = 0;
            *(candidates + i * N + j) = 0;
        }

    *(O + (N - 1) * N + N / 2) = 1;
}
```

Listing 1: Initialization

2..2 SOR Iteration

In this function, we respectively calculate the left, right, up, down value of each grid cell and then use equation (7) to update the concentration value.
The while loop stops when $delta <= tol$.

```
void SOR(float *C_old, float *C_current, int *O)
{
    float delta;
    float right, left, up, down;

    do
    {
        delta = 0;
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
            {
                if (*(O + i * N + j) == 1)
                    continue;

                left = (j == 0) ? *(C_current + i * N + (N - 1)) : *(C_current + i *
                    N + j - 1);
                right = (j == N - 1) ? *(C_current + i * N + 0) : *(C_current + i *
                    N + j + 1);
                up = (i == 0) ? c0 : *(C_current + (i - 1) * N + j);
                down = (i == N - 1) ? cN : *(C_current + (i + 1) * N + j);
                *(C_current + i * N + j) = 0.25 * omega * (left + right + up + down)
                    + (1 - omega) * *(C_current + i * N + j);
                delta = fmax(delta, fabs(*(C_current + i * N + j) - *(C_old + i * N
                    + j)));
            }
        // Update C_old
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                *(C_old + i * N + j) = *(C_current + i * N + j);

    } while (delta > tol);
}
```

Listing 2: SOR Iteration

2..3 Growth

Growing of the object requires three step:

1. Determine growth candidates;
2. Determine growth probabilities;
3. Grow.

For each growth candidate a random number between zero and one is drawn and if the random number is smaller than the growth probability, this specific site is successful and is added to the object. In this way, on average just one single site is added to the object. In our implementation, function **r2()** is used to generate a random number between zero and one.

```
void growth(float *C_current, int *O, int *candidates)
{
    int left, right, up, down;
```

```
    *nutri = 0;

    // reset candidates
    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            *(candidates + i * N + j) = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            if (*(O + i * N + j) == 1)
                continue;

            left = (j == 0) ? 0 : *(O + i * N + j - 1);
            right = (j == N - 1) ? 0 : *(O + i * N + j + 1);
            up = (i == 0) ? 0 : *(O + (i - 1) * N + j);
            down = (i == N - 1) ? 0 : *(O + (i + 1) * N + j);
            if ((left == 1 || right == 1 || up == 1 || down == 1))
            {
                *(candidates + i * N + j) = 1;
                *nutri += *(C_current + i * N + j);
            }
            else
                *(candidates + i * N + j) = 0;
        }

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            if ((*(candidates + i * N + j) == 1) && (r2() <= (*(C_current + i * N +
                j) / *nutri)))
            {
                *(O + i * N + j) = 1;
                *(C_current + i * N + j) = 0;
            }
}
```

Listing 3: Grow

3. Parallel Code

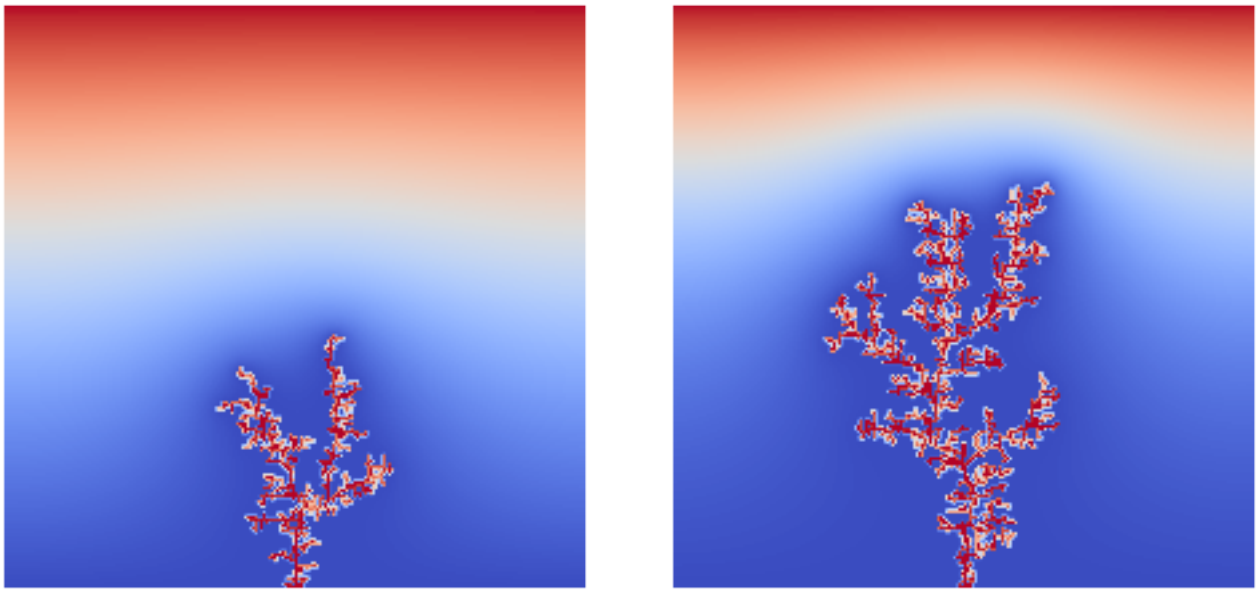
The parallel version is based on the serial version. So we only show the difference between them. The results is in Figure 2

3..1 Initialization

Here we initialize the grid for each CPU independently. Another way to initialize is initialize in the main CPU and scatter the grid to each CPU, but it is slower than our approach. The details implementation is in Appendix 2.

3..2 SOR Iteration

In the parallel version, we need to reorder the computations. First, color the computational grid as a checkerboard, with red and black grid points. Next, given the fact that the stencil



(a) 800 Iterations (b) 2000 Iterations

Figure 2: Results of Parallel DLA growth on a 200^2 lattice.

in the update procedure only extends to the nearest neighbors, it turns out that all red points are independent from each other (they only depend on black points) and vice-versa. Red and balck grid points respectively corresponse to $\mathbf{r} = 0$ and $\mathbf{r} = 1$ in our implementation. We also do the computation in place, and use new results as soon as they become available.

Here, we communicate array **C_below** and **C_above** for each CPU. Note that the size of each array is only $(N + 1)//2$ since it is not necessary to exchange the complete set of boundary points, but only half. To splits the first and last row of the grid part to send, we create a custom data type called **everytwice** using **MPI_Datatype** and **MPI_Type_vector**. This help us sending segments of an array with stride 2. We need additional value call **dieu_chinh** to ajust the begin of the **everytwice** structure based on **r** value. Details implementation can be found in the appendix 3..

```
MPI_Datatype everytwice;
MPI_Type_vector((N + 1) / 2, 1, 2, MPI_FLOAT, &everytwice);
MPI_Type_commit(&everytwice);
```

Listing 4: **everytwice** data type

```
dieu_chinh = (Np * rank + 1 + r) % 2;
```

Listing 5: **dieu_chinh**

3..3 Growth

In three steps of growth we have discussed in previous section:

1. The growth step is local, and therefore can be computed completely in parallel.
2. Calculation of the growth probabilities requires a global communication (for the normalization, i.e. the denominator of the equation for the probability)

To normalize the probabilities, we need to calculate the sum of the nutries in all the grid cells. We can use the **MPI_Allreduce** function to do this.

```
MPI_Allreduce(nutri, &global_nutri, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

Listing 6: Compute Global Nutries

Appendices

A Parallel Functions List

1. Main

```
    KhoiTao(C_old, C_current, O, candidates);

    for (k = 0; k < iter; k++)
    {
        SOR();
        for (i = 0; i < Np; ++i)
            for (j = 0; j < N; ++j)
                *(C_old + i * N + j) = *(C_current + i * N + j);
        growth();
    }

    float *C_current_global;
    int *O_global;
    // print
    if (rank == 0)
    {
        C_current_global = (float *)malloc(N * N * sizeof(float));
        O_global = (int *)malloc(N * N * sizeof(int));
    }
    MPI_Gather(C_current, Np * N, MPI_FLOAT, C_current_global, Np * N, MPI_FLOAT,
        0, MPI_COMM_WORLD);
    MPI_Gather(O, Np * N, MPI_INT, O_global, Np * N, MPI_INT, 0, MPI_COMM_WORLD);
```

Listing 7: Main Function

2. Initialization

```
void KhoiTao(float *C_old, float *C_current, int *O, int *candidates)
{
    // C Initialization
    for (i = 0; i < Np; ++i)
        for (j = 0; j < N; ++j)
        {
            *(C_current + i * N + j) = 0;
            *(C_old + i * N + j) = 0;
            *(O + i * N + j) = 0;
        }

    // Object Initialization
    if (rank == size - 1)
        *(O + (Np - 1) * N + N / 2) = 1;
}
```

Listing 8: Initialization Parallel

3. SOR Iteration

```
void SOR()
{
    float right, left, up, down;
    float global_alpha;
    int dieu_chinh;
    MPI_Datatype everytwice;
    MPI_Type_vector((N + 1) / 2, 1, 2, MPI_FLOAT, &everytwice);
    MPI_Type_commit(&everytwice);

    do
    {
        *alpha = 0;
        for (r = 0; r < 2; r++)
        {
            // He so cho tung Np khac nhau
            dieu_chinh = (Np * rank + 1 + r) % 2;

            // Send down
            if (rank != size - 1)
                MPI_Send(C_current + (Np - 1) * N + (dieu_chinh + Np) % 2, 1,
                    everytwice, rank + 1, 0, MPI_COMM_WORLD);

            // Send up
            if (rank != 0)
                MPI_Send(C_current + 1 - dieu_chinh, 1, everytwice, rank - 1, 1,
                    MPI_COMM_WORLD);

            // Receive from below
            if (rank != size - 1)
                MPI_Recv(C_below, (N + 1) / 2, MPI_FLOAT, rank + 1, 1,
                    MPI_COMM_WORLD, &status);
            else
            {
                for (i = 0; i < (N + 1) / 2; ++i)
                    C_below[i] = cN;
            }

            // Receive from above
            if (rank != 0)
                MPI_Recv(C_above, (N + 1) / 2, MPI_FLOAT, rank - 1, 0,
                    MPI_COMM_WORLD, &status);
            else
            {
                for (i = 0; i < (N + 1) / 2; ++i)
                    C_above[i] = c0;
            }

            for (i = 0; i < Np; i++)
            {
                for (j = 0; j < N; j++)
                {
```

```
                // ignore object position
                if (*(O + i * N + j) == 1)
                    continue;

                // ignore half of the grid
                if ((rank * Np + i + j) % 2 == r)
                    continue;

                left = (j == 0) ? *(C_current + i * N + (N - 1)) : *(C_current +
                    i * N + j - 1);
                right = (j == N - 1) ? *(C_current + i * N) : *(C_current + i * N
                    + j + 1);
                up = (i == 0) ? *(C_above + j / 2) : *(C_current + (i - 1) * N +
                    j);
                down = (i == Np - 1) ? *(C_below + j / 2) : *(C_current + (i + 1)
                    * N + j);

                *(C_current + i * N + j) = (1 - omega) * *(C_current + i * N + j)
                    + omega * (left + right + up + down) / 4;

                *alpha = fmax(*alpha, fabs(*(C_current + i * N + j) - *(C_old + i
                    * N + j)));
            }
        }
    }

    for (i = 0; i < Np; ++i)
        for (j = 0; j < N; ++j)
            *(C_old + i * N + j) = *(C_current + i * N + j);

    MPI_Allreduce(alpha, &global_alpha, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
} while (global_alpha > tol);

MPI_Type_free(&everytwice);
}
```

Listing 9: SOR Iteration

4. Growth

```
void growth()
{
    int left, right, up, down;
    float global_nutri;

    for (i = 0; i < N; ++i)
    {
        *(O_below + i) = 0;
        *(O_above + i) = 0;
    }

    // send the first row of object to above
```

```
if (rank != 0)
    MPI_Send(O, N, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);

// send the last row of object to below
if (rank != size - 1)
    MPI_Send(O + (Np - 1) * N, N, MPI_INT, rank + 1, 1, MPI_COMM_WORLD);

// receive object from below
if (rank != size - 1)
    MPI_Recv(O_below, N, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, &status);

// receive object from above
if (rank != 0)
    MPI_Recv(O_above, N, MPI_INT, rank - 1, 1, MPI_COMM_WORLD, &status);

// reset candidates
for (i = 0; i < Np; ++i)
    for (j = 0; j < N; ++j)
        *(candidates + i * N + j) = 0;

// reset nutri
*nutri = 0;

// calculate candidates and nutri
for (i = 0; i < Np; i++)
{
    for (j = 0; j < N; j++)
    {
        if (*(O + i * N + j) == 1)
        {
            continue;
        }

        left = (j == 0) ? 0 : *(O + i * N + j - 1);
        right = (j == N - 1) ? 0 : *(O + i * N + j + 1);
        up = (i == 0) ? *(O_above + j) : *(O + (i - 1) * N + j);
        down = (i == Np - 1) ? *(O_below + j) : *(O + (i + 1) * N + j);

        if ((left == 1 || right == 1 || up == 1 || down == 1))
        {
            *(candidates + i * N + j) = 1;
            *nutri += *(C_current + i * N + j);
        }
        else
            *(candidates + i * N + j) = 0;
    }
}

// Reduce
MPI_Allreduce(nutri, &global_nutri, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);

for (i = 0; i < Np; ++i)
    for (j = 0; j < N; ++j)
```

```
            if (*(candidates + i * N + j) == 1 && r2() <= (*(C_current + i * N + j)
                / global_nutri))
            {
                *(O + i * N + j) = 1;
                *(C_current + i * N + j) = 0;
            }
}
```

Listing 10: Growth