Project Report

Diffusion Limited Aggregation

Course: IT4130E - Parallel and Distributed Programming

Authors: GROUP 02

Nguyen Tong Minh - 20204885
Nguyen Xuan Thai Hoa - 20204878
Ly Nhat Nam - 20204886
Nguyen Cong Dat - 20200137
Hoang Xuan Viet - 20200662

Advisor: Dr. Vu Van Thieu

Academic semester: 2022.2

Abstract

Diffusion Limited Aggregation is a generative model inspired by diffusion and aggregation of zinc ions in physics and chemistry field. The model divides into two main steps: time-independent diffusion and growth of the object. In this report we mainly focus on the first step by implementing three parallel iteration algorithms: Jacobi Iterative Method, Gauss-Seidel Iterative Method and the Successive Over Relaxation(SOR) that inherit from both predecessor. The detailed information of the implementation and its result are also provided in the report.

Contents

1 Problem Formulation **3**

2 Algorithms/Parallel Designs **4**
2.1 Jacobi Iterative Method 4
2.2 Gauss-Seidel Iterative Method 5
2.3 Successive Over Relaxation 7

3 Experiments **8**
3.1 Implementation Details 8
3.2 Results 8

1 Problem Formulation

Many attractive images and life-like structures can be generated using models of physical processes from areas of chemistry and physics. One such example is diffusion limited aggregation or DLA which describes, among other things, the diffusion and aggregation of zinc ions in an electrolytic solution onto electrodes. "Diffusion" because the particles forming the structure wander around randomly before attaching themselves ("Aggregating") to the structure. "Diffusion-limited" because the particles are considered to be in low concentrations so they don't come in contact with each other and the structure grows one particle at a time rather then by chunks of particles. Other examples can be found in coral growth, the path taken by lightning, coalescing of dust or smoke particles, and the growth of some crystals. Perhaps the first serious study of such processes was made by Witten, T.A. and Sander, L. M. and published by them in 1981 [1].

For example, DLA can model a Bacilus 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 shown in algorithm 1.

Algorithm 1 DLA algorithm

time-independent diffusion eq.: $\Delta c = 0$

loop till convergence:

1. Solve $\Delta c = 0$. Object be sink ($c = 0$).
2. Let object grow.

The first step is done by a parallel SOR iteration. The next step, growth of the object, requires three sub-steps: Determine growth candidates; determine growth probabilities and grow.

A growth candidate is basically a lattice site that is not part of the object, but whose north -, or east -, or south -, or west neighbor is part of the object. In figure 1, a possible configuration of the object is shown; the black circles form the current object, while the white circles are the growth candidates.

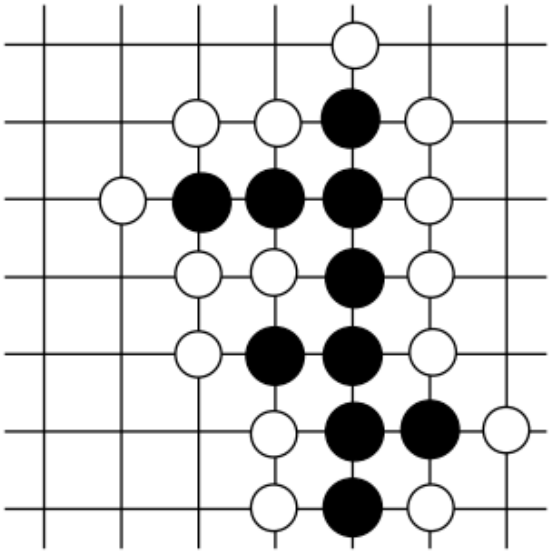

Figure 1: The object and possible growth sites.

The probability for growth at each of the growth candidates is calculated by:

$$p_g\left((i,j) \in \circ \rightarrow (i,j) \in \bullet\right) = \frac{\left(c_{i,j}\right)^{\eta}}{\sum_{(i,j)\in\circ}\left(c_{i,j}\right)^{\eta}}.$$

Figure 2: Growth probability at a candidate.

The parameter η determines the shape of the object. For $\eta = 1$ we get the normal DLA cluster, i.e., a fractal object. For $\eta < 1$ the object becomes more compact (with $\eta = 0$ resulting in the Eden cluster), and for $\eta > 1$ the cluster becomes more open (and finally resembles say a lightning flash).

Modelling the growth is now a simple procedure. 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.

Time-independent diffusion Let us consider the two-dimensional time-dependent diffusion equation:

$$\frac{\partial c}{\partial t} = D(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}) \tag{1}$$

where the concentration c is a function of spatial variables x, y and time t. In many cases, one is only interested in the final steady state concentration field and not so much in the transient behaviour, 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 (i.e., diffusion limited aggregation).

Setting all time derivatives to zero in the diffusion equation 1, we find the time-independent diffusion equation:

$$\Delta c = 0 \tag{2}$$

This is the famous Laplace equation that is encountered in many places in science and engineering, and is therefore very relevant to study in some more detail.

Indeed, again consider the two-dimensional domain and the discretization. The concentration on the grid point here as $c_{l,m}$. Substituting the finite difference formulation for the spatial derivatives into the equation 2 results in the time-independent diffusion equation 3.

$$c_{l,m} = \frac{1}{4}[c_{l-1,m} + c_{l+1,m} + c_{l,m-1} + c_{l,m+1}], \forall (l,m) \tag{3}$$

Grid conditions As in our problem, we are facing with 2D grid. Then we have a strictly boundary simulation:

- Periodic in columns direction, with a period equals to N (width of the grid).
- Constant in row direction.

The initial object we can put anywhere on the grid, and this can be setting by change the source code.

2 Algorithms/Parallel Designs

2.1 Jacobi Iterative Method

Since n is the iteration index, this iterative scheme is known as the Jacobi iteration. Take note of the relationship between the above equation and the finite difference scheme for the time dependent diffusion equation. If we take the by the CFL condition that allow maximum time step, i.e. $\frac{\delta x^2}{\delta t} = 1/4$, we reproduce the Jacobi iteration of above equation. In other words, we anticipate that the Jacobi iteration will suffer from the relatively large number of iterations required before convergence.

The code for a Jacobi iteration's inner loops is provided by algorithm 2. Once more, we'll be working with a square domain that has fixed borders in the y direction and periodic boundaries in the x direction. Additionally, a certain stopping criterion is used. We demand that

$$\max |c_{ij}^{(n+1)} - c_{ij}^{(n)}| < \varepsilon \tag{4}$$

The difference in concentration between two iterations on all grid points should be smaller that some small number ε. This is more or less a serve halting circumstance. There are more methods that can be employed, such as computing the mean difference and requiring that it be less than a specific tiny number. The halting criterion will not be discussed further

here, but it should be understood as an important issue to be taken into account in any new applications of an iterative approach.

```
/* Jacobi update, square domain, periodic in x, fixed */
/* upper and lower boundaries                          */
do {
   δ = 0
   for i=0 to max {
      for j=0 to max {
         if(c_ij is a source) c_ij^(n+1) = 1.0
         else if(c_ij is a sink) c_ij^(n+1) = 0.0
         else {
            /* periodic boundaries */
            west = (i==0) ? c_max-1,j^(n) : c_i-1,j^(n)
            east = (i==max) ? c_1,j^(n) : c_i+1,j^(n)
            /* fixed boundaries */
            south = (j==0) ? c0 : c_i,j-1^(n)
            north = (j==max) ? cL : c_i,j+1^(n)
             c_ij^(n+1) = 0.25 * (west + east + south + north)
         }
         /* stopping criterion */
         if(|c_ij^(n+1) - c_ij^(n)| > tolerance) δ = |c_ij^(n+1) - c_ij^(n)|
      }
   }
while (δ > tolerance)
```

Figure 3: The sequential Jacobi iteration.

At first sight a parallel Jacobi iteration seems very straightforward. The computation is again based on a local five-point stencil, as in the time dependent case. Therefore, we can apply a domain decomposition, and resolve all dependencies by first exchanging all boundary points, followed by the update using the Jacobi iteration. However, the main new issue is the stopping condition. In the time dependent case we know before run time how many time steps need to be taken. Therefore each processor knows in advance how many iterations are needed. In the case of the Jacobi iteration (and all other iterative methods) we can only decide when to stop during the iterations. We must therefore implement a parallel stopping criterion. This posses a problem, because the stopping condition is based on some global measure (e.g. a maximum or mean error over the complete grid). That means that a global communication (using e.g. `MPI-Reduce`) is needed that may induce a large communication overhead. In practical implementations one must seriously think about this. Maybe it pays off to calculate the stopping condition once every q iterations (with q some small positive number) instead of after each iteration. With all this in mind the pseudo code for the parallel Jacobi iteration is given in algorithm 4. It is clear that it closely resembles the time-dependent pseudo code.

```
main () /* pseudo code for parallel Jacobi iteration */
{
   decompose lattice;
   initialize lattice sites;
   set boundary conditions;
   do {
      exchange boundary strips with neighboring processors;
      for all grid points in this processor {
         update according to Jacobi iteration;
         calculate local δ_i parameter; /* stopping criterion */
      }
      obtain the global maximum δ of all local δ_i values
   }
   while (δ > tolerance)
   print results to file;
}
```

Figure 4: Pseudo code for the parallel Jacobi iteration.

2.2 Gauss-Seidel Iterative Method

The Jacobi iteration is not very efficient, and here we will introduce a first step to improve the method. The Gauss-Seidel iteration is obtained by applying a simple idea. In the Jacobi iteration we always use results from the previous iteration to update a point, even when we already have new results available. The idea of Gauss-Seidel iteration is to apply new results as soon as they become available. In order to write down a formula for the Gauss-Seidel iteration we must specify the order in which we update the grid points. Assuming a row-wise update procedure (i.e. we increment l while keeping m fixed) we find for the Gauss-Seidel iteration.

$$c_{l,m}^{n+1} = \frac{1}{4}[c_{l-1,m}^{(n)} + c_{l+1,m}^{(n+1)} + c_{l,m-1}^{(n)} + c_{l,m+1}^{(n+1)}] \quad (5)$$

One immediate advantage of the Gauss-

Seidel iteration lies in the memory usage. In the Jacobi iteration you would need two arrays, one to store the old results, and another to store the new results. In Gauss-Seidel you immediately use the new results as soon as they are available. So, we only need one array to store the results. Especially for large grids this can amount to enormous savings in memory! We say that the Gauss-Seidel iteration can be computed in place.

Is Gauss-Seidel iteration also faster than Jacobi iteration. According to theory it turns out that a Gauss-Seidel iteration requires a factor of two iterations less then Jacobi. However, The Gauss-Seidel iteration posses a next challenge to parallel computation. At first sight we must conclude that the parallelism available in Jacobi iteration is now completely destroyed by the Gauss-Seidel iteration. Gauss-Seidel iteration seems inherently sequential. Well, it is in the way we introduced it, with the row-wise ordering of the computations. However, this row-wise ordering was just a convenient choice. It turns out that if we take another ordering of the computations we can restore parallelism in the Gauss-Seidel iteration. This is an interesting case where reordering of computations provides parallelism.

The idea of the reordering of the computations is as follows. First, color the computational grid as a checkerboard, with red and black grid points. Next, given the fact that the stencil 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. So, instead of the row-wise ordering we could do a red-black ordering, were we first update all red points, and next the black points (see figure 5). We also call this Gauss-Seidel iteration, because although the order in which grid points are updated is now different, we also do the computation in place, and use new results as soon as they become available.

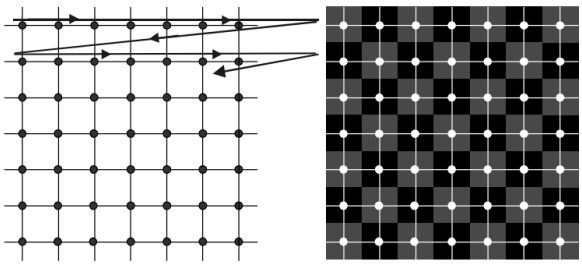

Figure 5: Row wise ordering (left) versus red-black ordering (right).

```
/* only the inner loop of the parallel Gauss-Seidel method with */
/* Red Black ordering */
   do {
      exchange boundary strips with neighboring processors;
      for all red grid points in this processor {
         update according to Gauss-Seidel iteration;
      }
      exchange boundary strips with neighboring processors;
      for all black grid points in this processor {
         update according to Gauss-Seidel iteration;
      }
      obtain the global maximum δ of all local δ_i values
   }
   while (δ > tolerance)
```

Figure 6: The pseudo code for parallel Gauss-Seidel iteration with red-black ordering.

This new red-black ordering restores parallelism. We can now first update all red points in parallel, followed by a parallel update of all black point. The pseudo code for parallel Gauss-Seidel with red-black ordering is show in algorithm 6. The computation is now split into two parts, and before each part a communication with neighboring processors is needed. On first sight this would suggest that the parallel Gauss-Seidel iteration requires twice as many communication time in the exchange part. This however is not true, because it is not necessary to exchange the complete set of boundary points, but only half. This is because for updating red points we only need the black point, so also only the black boundary points need to be exchanged. This means that, in comparison with parallel Jacobi, we only double the setup times required for the ex-

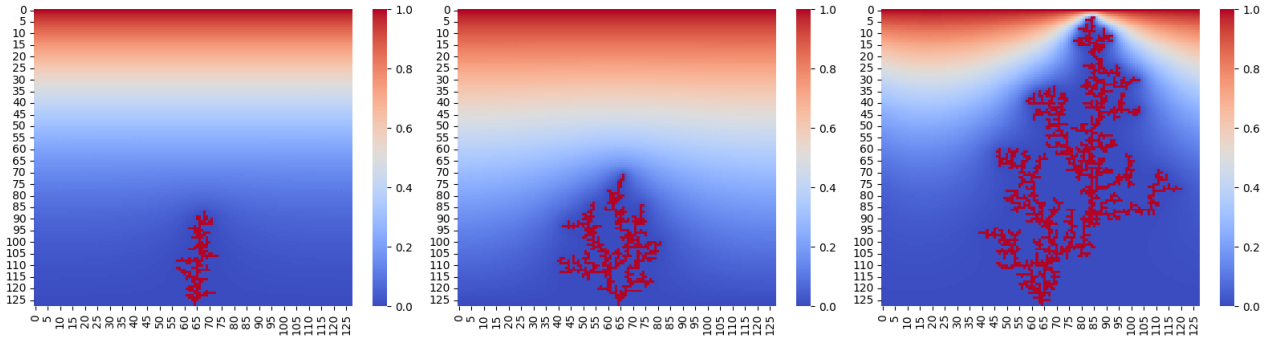

Figure 7: Number of iterations: 250, 300, 1500, respectively.

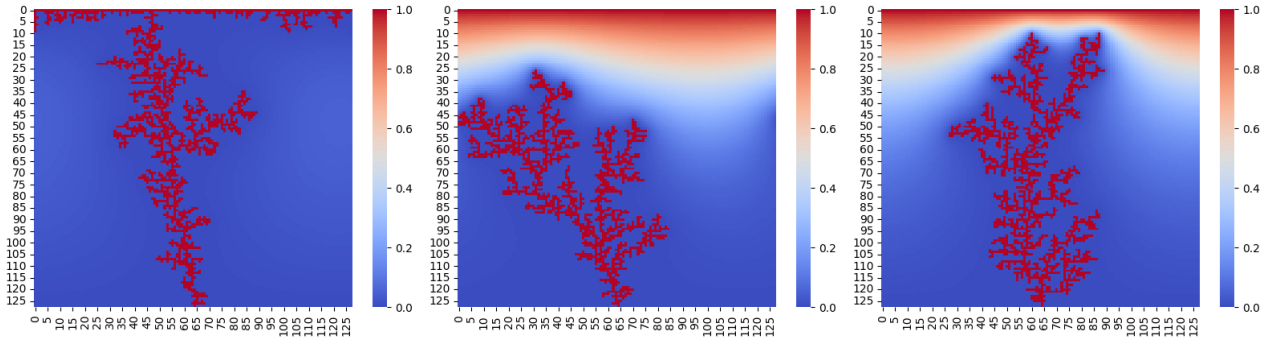

Figure 8: $\omega = 1.3, 1.6, 1.9$, respectively.

change operations, but keep the sending times constant. Finally note that the same parallel stop condition as before can be applied.

2.3 Successive Over Relaxation

The ultimate step in the series from Jacobi and Gauss-Seidel is to apply a final and as will become clear very efficient idea. In the Gauss-Seidel iteration the new iteration results is completely determined by its four neighbors. In the final method we apply a correction to that, by mixing the Gauss-Seidel result with the current value, i.e.

$$c_{l,m}^{n+1} = \frac{\omega}{4}[c_{l-1,m}^{(n)} + c_{l+1,m}^{(n+1)} + c_{l,m-1}^{(n)} + c_{l,m+1}^{(n+1)}] + (1-\omega)c_{l,m}^{(n)} \quad (6)$$

The parameter co determines the strength of the mixing. One can prove that for $0 < \omega < 2$ the method is convergent. For $\omega = 1$ we recover the Gauss-Seidel iteration, for $0 < \omega < 1$ the method is called Successive Under Relaxation. For $1 < \omega < 2$ we speak of Successive Over Relaxation, or SOR.

It turns out that for finite difference schemes SOR is very advantageous and gives much faster convergence then Gauss-Seidel. The number of needed iterations will however strongly depend on the value of ω. Following the instructors' documents, ω is taken close to 1.9. Finally note that parallel SOR is identical to parallel Gauss-Seidel. Only the update rules have been changed.

By a way, we can consider that SOR is Gaussian-Sidel with momentum mechanism enabling better guide of expansion.

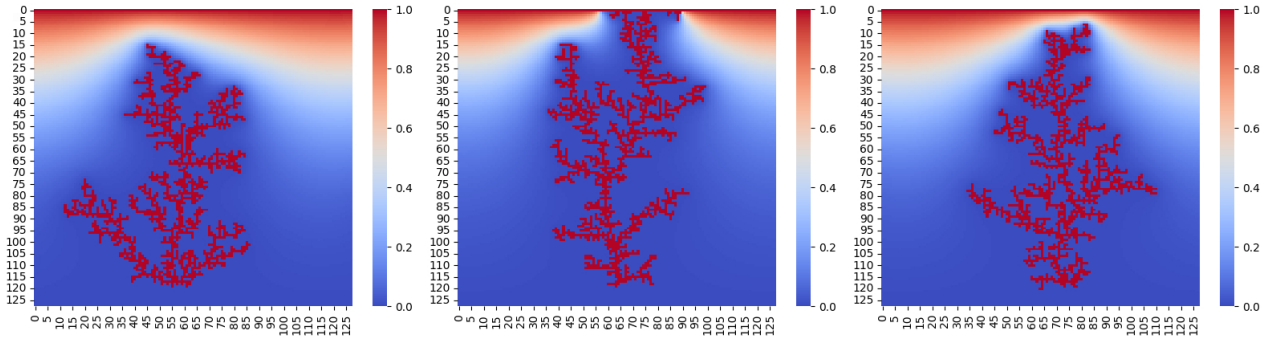

Figure 9: Initialize at middle, middle-left, middle-right of the grid, respectively.

3 Experiments

3.1 Implementation Details

The input grid is $N \times N$ (i.e., $N = 128$). We divide it into n_p processes one of which handles $n_r = \frac{N}{n_p}$ rows of the grid. The task of each processor is passing message to get information from the boundary, estimating the diffusion process in each cell and then exporting data.

Setup The SOR iterative method with red-black ordering is implemented using C programming language with Message Passing Interface (MPI) for parallel mechanisms.

3.2 Results

We carry the experiments by varying the number of iterations, object initialization locations and the values of ω, that constitute different initialization schemas, as shown in figures 7 and 8, 9.

References

[1] T. A. Witten and L. M. Sander. "Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon". In: *Phys. Rev. Lett.* 47 (19 Nov. 1981), pp. 1400–1403. DOI: 10.1103/PhysRevLett.47.1400. URL: https://link.aps.org/doi/10.1103/PhysRevLett.47.1400.