

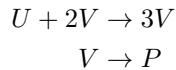
# ADI for Reaction-Diffusion Systems

Stephanie Christ

August 4, 2015

## 1 Gray-Scott Reaction-Diffusion System

Reaction-Diffusion systems describe how the concentration of chemical substances changes in space. The interaction of the two chemical species  $U$  and  $V$  is through chemical reactions, the distribution in space due to diffusion. The reaction between the two species can be described by the following chemical terms:



By combining the reaction and the diffusion, we obtain differential equations for the concentrations  $u$  and  $v$ .

$$\frac{\partial u}{\partial t} = D_u \Delta u - uv^2 + F(1-u) \quad (1)$$

$$\frac{\partial v}{\partial t} = D_v \Delta v + uv^2 - (F+k)v \quad (2)$$

$D_u$  and  $D_v$  denote the diffusion coefficients for  $u$  and  $v$  respectively.  $F$  and  $k$  are constant coefficient which describe at which rate the chemical reactions occur.

The model is in 2 dimensions.

## 2 Numerical Method

We split our two-dimensional domain in square cells.

To solve the equations for  $u$  and  $v$  (equations 1 and 2), we split the reaction and the diffusion parts to solve them separately. First, we solve the diffusion and then we add the reaction to the obtained result.  $u$  and  $v$  can both be computed separately in each time step, as their contributions to the other equation is only explicit.

### 2.1 Alternating Direction Implicit (ADI) method

To solve the diffusion equation  $\frac{\partial \rho}{\partial t} = D_\rho \Delta \rho$ , we use the ADI method. This method has first been proposed by Craig and Sneyd in 1988 [1].

First half-step:

$$\rho_{i,j}^{n+\frac{1}{2}} = \rho_{i,j}^n + \frac{D_\rho \delta t}{2} \left[ \frac{\partial^2 \rho_{i,j}^{n+\frac{1}{2}}}{\partial x^2} + \frac{\partial^2 \rho_{i,j}^n}{\partial y^2} \right] \quad (3)$$

Second half-step:

$$\rho_{i,j}^{n+1} = \rho_{i,j}^{n+\frac{1}{2}} + \frac{D_\rho \delta t}{2} \left[ \frac{\partial^2 \rho_{i,j}^{n+\frac{1}{2}}}{\partial x^2} + \frac{\partial^2 \rho_{i,j}^{n+1}}{\partial y^2} \right] \quad (4)$$

We approximate the second derivative using a second order central differences scheme:

$$\frac{\partial \rho_{i,j}}{\partial x^2} = \frac{\rho_{i-1,j} - 2\rho_{i,j} + \rho_{i+1,j}}{\Delta x^2} \quad (5)$$

$$\frac{\partial \rho_{i,j}}{\partial y^2} = \frac{\rho_{i,j-1} - 2\rho_{i,j} + \rho_{i,j+1}}{\Delta y^2} \quad (6)$$

Inserting this into equation 3 for the first half-step and setting  $\Delta x = \Delta y$ , we obtain the following equation:

$$\rho_{i,j}^{n+\frac{1}{2}} = \rho_{i,j}^n + \frac{D_\rho \delta t}{2} \left[ \frac{\rho_{i-1,j}^{n+\frac{1}{2}} - 2\rho_{i,j}^{n+\frac{1}{2}} + \rho_{i+1,j}^{n+\frac{1}{2}}}{\Delta x^2} + \frac{\rho_{i-1,j}^n - 2\rho_{i,j}^n + \rho_{i+1,j}^n}{\Delta x^2} \right] \quad (7)$$

We sort the implicit and explicit parts and get the following system:

$$\begin{aligned} & -\frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i-1,j}^{n+\frac{1}{2}} + \left( 1 + \frac{D_\rho \Delta t}{\Delta x^2} \right) \rho_{i,j}^{n+\frac{1}{2}} - \frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i+1,j}^{n+\frac{1}{2}} \\ & = \frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i,j-1}^n + \left( 1 - \frac{D_\rho \Delta t}{\Delta x^2} \right) \rho_{i,j}^n - \frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i,j+1}^n \end{aligned} \quad (8)$$

This gives us a tridiagonal system with a constant on each diagonal for each row of the domain. On the lower and upper diagonals, we have the coefficient  $-\frac{D_\rho \Delta t}{2\Delta x^2}$ , on the middle diagonal the coefficient  $\left( 1 + \frac{D_\rho \Delta t}{\Delta x^2} \right)$ .

Performing analogous calculations for the second half-step 4 yields

$$\begin{aligned} & -\frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i,j-1}^{n+1} + \left( 1 + \frac{D_\rho \Delta t}{\Delta x^2} \right) \rho_{i,j}^{n+1} - \frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i,j+1}^{n+1} \\ & = \frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i-1,j}^{n+\frac{1}{2}} + \left( 1 - \frac{D_\rho \Delta t}{\Delta x^2} \right) \rho_{i,j}^{n+\frac{1}{2}} - \frac{D_\rho \Delta t}{2\Delta x^2} \rho_{i+1,j}^{n+\frac{1}{2}} \end{aligned} \quad (9)$$

The second half-step gives a tridiagonal system to solve for each column of the domain.

### 2.1.1 Thomas Algorithm

The Thomas Algorithm – named after Llewellyn Thomas – is an algorithm to solve tridiagonal systems  $Ax = d$  in  $\mathcal{O}(n)$  time. It is a form of Gaussian

Elimination.

$$Ax = d \quad (10)$$

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} \quad (11)$$

The method first modifies the coefficients in the following way:

$$c'_i = \begin{cases} \frac{c_i}{b_i} & ; \quad i = 0 \\ \frac{c_i}{b_i - a_i c'_{i-1}} & ; \quad i = 1, 2, \dots, n-1 \end{cases} \quad (12)$$

$$d'_i = \begin{cases} \frac{d_i}{b_i} & ; \quad i = 0 \\ \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & ; \quad i = 1, 2, \dots, n. \end{cases} \quad (13)$$

Then the solution is obtained by backwards substitution.

$$x_n = d'_n \quad (14)$$

$$x_i = d'_i - c'_i x_{i+1} \quad ; \quad i = n-1, n-2, \dots, 0. \quad (15)$$

## 2.2 Reaction Term

To add the reaction term to the solution of the diffusion equation, we use a forward Euler scheme.  $\tilde{u}^{n+1}$  and  $\tilde{v}^{n+1}$  denote the solution of the diffusion equations before adding the contribution of the reaction term.

$$u_{i,j}^{n+1} = \tilde{u}_{i,j}^{n+1} + \Delta t (-\tilde{u}_{i,j}^{n+1}(\tilde{v}_{i,j}^{n+1})^2 + F(1 - \tilde{u}_{i,j}^{n+1})) \quad (16)$$

$$v_{i,j}^{n+1} = \tilde{v}_{i,j}^{n+1} + \Delta t (\tilde{u}_{i,j}^{n+1}(\tilde{v}_{i,j}^{n+1})^2 - (F + k)\tilde{v}_{i,j}^{n+1}) \quad (17)$$

## 2.3 Boundary conditions

We use reflecting boundary conditions. This modifies the tridiagonal matrix system, so that the values on each diagonal aren't constant for the whole diagonal anymore. The first value on the upper diagonal, as well as the last value on the lower diagonal, are doubled.

## 3 Implementation

We save the grid for  $u$  and  $v$  each in a `std::vector<double>` in row-wise storage. For the tridiagonal matrix, we wrote our own class to simplify the access to the diagonals.

```

1 // loop over all rows
2
3 // j=0
4 for (int i=0; i<N_; ++i) {
5     uRhs[i] = U(i,0) + uCoeff * (U(i,1) - U(i,0));
6 }
7 TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,0), 1);
8
9 // inner grid points
10 for (int j=1; j<N_-1; ++j) {
11     // create right-hand side of the systems
12     for (int i=0; i<N_; ++i) {
13         uRhs[i] = U(i,j) + uCoeff * (U(i,j+1) - 2.*U(i,j) + U(i, ←
14             j-1));
15     }
16     TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,j), 1);
17 }
18
19 // j=N_-1
20 for (int i=0; i<N_; ++i) {
21     uRhs[i] = U(i,N_-1) + uCoeff * (- U(i,N_-1) + U(i,N_-2));
22 }
23 TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,N_-1), 1);

```

Listing 1: Serial implementation of the first half-step (the computations for  $v$  are omitted as they are analogue to the computation for  $u$ )

To visualize the result, we used OpenGL for displaying the concentration of  $u$  during the computation, and LodePNG<sup>1</sup> – a PNG image encoder and decoder by Lode Vandevenne – to save the image after the simulation.

### 3.1 Serial version

We loop over all time steps and apply the numerical scheme described above.

In each step, we first apply the ADI method to  $u$  and  $v$  to solve the diffusion part of the problem. Then we solve the reaction.

For the ADI-step, we have to treat the boundary rows/columns differently, as the dependencies on the neighbouring cells change (see 1. To facilitate the access to a grid cell, we have defined macros that let us use the two-dimensional coordinates of the cells. The solver for the tridiagonal matrix is an implementation of the Thomas algorithm, see section 3.2.

The second half-step is implemented analogously to the first half-step, but with the outermost loop over the columns  $i$  instead of the rows  $j$ .

For the reaction, we used a simple loop over all grid cells and add the reaction term to  $u$  and  $v$  (see listing 2).

```

1 double tmp;
2 for (int i=0; i<N_; ++i) {
3     for (int j=0; j<N_; ++j) {
4         tmp = U(i,j);

```

<sup>1</sup><http://lodev.org/lodepng/>

---

```

1 namespace TriDiagMatrixSolver
2 {
3     /**
4      * Solve a tridiagonal matrix system.
5      *
6      * @param n           number of elements in the result
7      * @param mat         tridiagonal matrix
8      * @param rhs         right-hand side of the system
9      * @param result      vector for the result, pointer to the ←
10     first element
11     * @param inc         increment for the elements of the result
12     */
13     void solve(unsigned int n, const TriDiagMatrix& mat, const ←
14             std::vector<double>& rhs, double *result, const unsigned ←
15             int inc);
16 }

```

---

Listing 3: Declaration of our implementation of the Thomas algorithm

---

```

5     U(i,j) += dt_ * ( -tmp*V(i,j)*V(i,j) + F_*(1.-tmp) );
6     V(i,j) += dt_ * ( tmp*V(i,j)*V(i,j) - (F_+k_-)*V(i,j) );
7 }
8 }

```

---

Listing 2: Reaction, serial implementaion

### 3.2 Thomas Algorithm

For the implementation of the Thomas Algorithm, we used the version on wikipedia from April 2014 [3]. We modified it so that we can pass the whole grid as an argument. Since the access to the grid is row-/column-wise, the argument **result** is a pointer to the first element we want to access. The increment **inc** is the increment we need to access the next element in the result. It is different for row-wise and column-wise access (e.g. first or second half-step), either 1 or  $N$  (the size of the grid in one dimension).

### 3.3 Parallelization

#### 3.3.1 OpenMP

First, we parallelized our code using OpenMP. In the diffusion part, we parallelized the outer **for**-loop (over  $j$  in the listing 1), since there are only read-dependencies between different  $j$ . The parallelization is according to listing 4.

The boundary computations aren't parallelized. We do not need to declare anything private, because each thread declares its own right-hand side, and all other variables have either read-only access or are accessed in different regions by each thread.

In the reaction part, we also parallelized the outer loop with a **#pragma omp parallel for**

```

1 #pragma omp parallel num_threads(nthreads_)
2 { // PARALLEL REGION BEGIN
3
4 // right hand sides for u and for v
5 std::vector<double> puRhs(N_);
6 std::vector<double> pvRhs(N_);
7
8 #pragma omp for
9 for (int j=1; j<N_-1; ++j) {
10     // row-wise computations, see listing 1
11 }
12 } // PARALLEL REGION END

```

Listing 4: Parallel OpenMP region

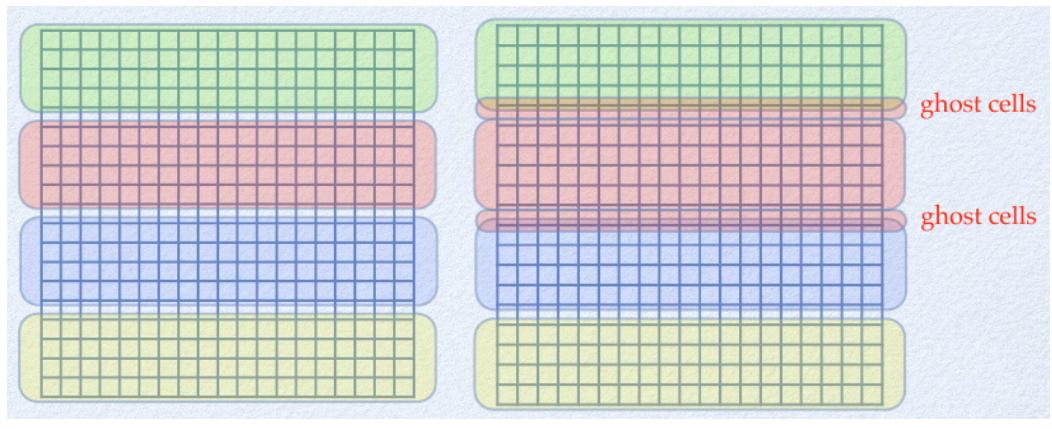


Figure 1: Domain decomposition for 4 MPI processes<sup>2</sup>

### 3.3.2 MPI

**Domain decomposition** For the MPI version, we decompose the domain and distribute it between the processes. This was done in blocks of rows, see figure 1a. To be able to distribute the rows equally, we changed the size of the domain to a multiple of the number of processors. As the first and last row need to access the values from the rows above/below, we add ghost rows (see figure 1b). The ghost rows don't get written to, but read, and need to be exchanged between processes before the computations. The grid is initialized on the different processes, so that each process has a vector of size  $N \times (N_{loc} + 2)$  for its part of the grid. The first and last row are for the ghost cells, so the processes only initialize  $N \times N_{loc}$  elements.

**Communication** The ghost cells are sent/received by the processes as shown in listing 5. We used nonblocking communication, because that lets us update the local inner grid cells (independent of ghost cells), while we wait on the

---

<sup>2</sup>Figures from HPCSE slides

```

1 MPI_Request request[8];
2 MPI_Status status[8];
3
4 // exchange boundaries
5 if (world.coord_x % 2 == 0) {
6
7     MPI_Isend(&u_[(Nx_loc)*(Ny_loc)], 1, bottom_boundary, world<-->
8             .bottom_proc, TAG, cart_comm, &request[0]);
9     MPI_Irecv(&u_[(Nx_loc+1)*(Ny_loc)], 1, bottom_boundary, world<-->
10            .bottom_proc, TAG, cart_comm, &request[1]);
11    MPI_Isend(&u_[(Ny_loc)], 1, top_boundary, world<-->
12            .top_proc, TAG, cart_comm, &request[2]);
13    MPI_Irecv(&u_[0], 1, top_boundary, world<-->
14            .top_proc, TAG, cart_comm, &request[3]);
15 }
16 else {
17
18     MPI_Irecv(&u_[0], 1, top_boundary, world<-->
19             .top_proc, TAG, cart_comm, &request[0]);
20     MPI_Isend(&u_[(Ny_loc)], 1, top_boundary, world<-->
21             .top_proc, TAG, cart_comm, &request[1]);
22     MPI_Irecv(&u_[(Nx_loc+1)*(Ny_loc)], 1, bottom_boundary, world<-->
23             .bottom_proc, TAG, cart_comm, &request[2]);
24     MPI_Isend(&u_[(Nx_loc)*(Ny_loc)], 1, bottom_boundary, world<-->
25             .bottom_proc, TAG, cart_comm, &request[3]);
26 }

```

Listing 5: Communication of the ghost cells

---

```

1 // send/receive ghost cells (non-blocking)
2
3 // update inner grid points (see listing 1)
4
5 // wait for boundaries to arrive
6 MPI_Waitall(8,request,status);
7
8
9 // update local boundaries
10
11 if (world.rank == 0) {
12     // i=0 local and global
13     for (int j=0; j<N_; ++j) {
14         uRhs[j] = U(0,j) + uCoeff * (U(1,j) - U(0,j));
15     }
16     TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(0,0), 1);
17 }
18 else {
19     // i=0 local
20     for (int j=0; j<N_; ++j) {
21         uRhs[j] = U(0,j) + uCoeff * (U(0+1,j) - 2.*U(0,j) + U<-
22             (0-1,j));
23     }
24     TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(0,0), 1);
25 }
26
27 if (world.rank == world.size-1) {
28     // i=Nx_loc-1 local and i=N_-1 global
29     for (int j=0; j<N_; ++j) {
30         uRhs[j] = U(Nx_loc-1,j) + uCoeff * (- U(Nx_loc-1,j) + U(<-
31             Nx_loc-2,j));
32     }
33     TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(Nx_loc<-
34             -1,0), 1);
35 }
36 else {
37     // i=Nx_loc-1 local
38     for (int j=0; j<N_; ++j) {
39         uRhs[j] = U(Nx_loc-1,j) + uCoeff * (U(Nx_loc-1+1,j) - 2.*<-
40             U(Nx_loc-1,j) + U(Nx_loc-1-1,j));
41     }
42     TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(Nx_loc<-
43             -1,0), 1);
44 }

```

---

Listing 6: First half-step parallelized with MPI

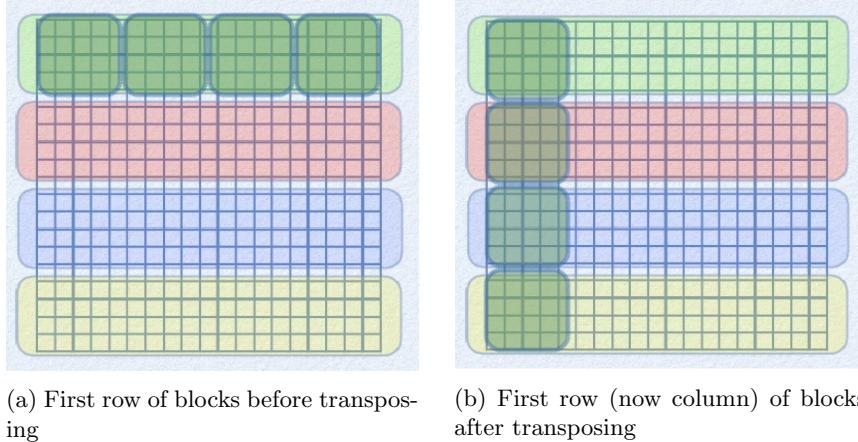


Figure 2: Domain decomposition for MPI transpose<sup>3</sup>

ghost cells to arrive. After updating the inner grid cells, we need to wait for the boundaries to arrive, then we can continue to update the boundary cells. We distinguish between rows that are only local boundaries, and those that are also global boundaries (first and last row), because they need to handle the reflecting boundary conditions correctly.

**Transposing the grid** After the first half-step, we need to transpose the grid, so that instead of the columns like in the serial version, we can work again on rows that are distributed between processes. For the transposing, we have implemented two different versions. Both versions first split the rows into blocks, such that each processor has a row of blocks. See figure 2 for a visualization of the blocks (all colours are split into blocks analogue to the green blocks that are shown). The processes send the blocks to another process using `MPI_Alltoall`. The first version then transposes each block locally (see listing 7). The second version uses a different receive data type than send data type. The receiving data type is ordered in a way that is the transpose of the send data type (see listing 8). This lets us skip the local transpose, as the received data is already transposed. We compare the performance of these two versions in section ??.

After transposing the grid, the second half-step is identical to the first half-step.

The reaction part of the simulation isn't changed from the serial implementation, except that the loop goes only over the local grid.

**MPI Data types** We defined four different MPI data types. Two are for the ghost cells (top and bottom rows), they are both identical and only defined twice for code readability (see listing 9). These data types are contiguous vectors. The other two are to transpose the grid (see listing 10). They are square blocks that split the blocks of rows each processor has. `block_send` and `block_recv` already contain the data we need to send/receive in the correct

---

<sup>3</sup>Figures adapted from HPCSE slides

```

1 MPI_Alltoall(&uTemp[Ny_loc], 1, block_resized_send, &u_[Ny_loc], ←
2   1, block_resized_send, MPI_COMM_WORLD);
3 // locally transpose blocks
4 int ind1, ind2;
5 // loop over blocks
6 for (int b=0; b<Nb_loc; ++b) {
7   for (int i=0; i<Nx_loc; ++i) {
8     for (int j=0; j<i; ++j) {
9       ind1 = (i+1)*Ny_loc + j + b*Nx_loc; // regular index ←
10      + offset of block
11      ind2 = (j+1)*Ny_loc + i + b*Nx_loc; // switch i and j
12      std::swap(u_[ind1], u_[ind2]);
13    }
14  }
15 }
```

Listing 7: Transposing the grid using MPI Alltoall and local transpose

```

1 MPI_Alltoall(&uTemp[Ny_loc], 1, block_resized_send, &u_[Ny_loc], ←
2   1, block_resized_recv, MPI_COMM_WORLD);
```

Listing 8: Transposing the grid using MPI Alltoall and different send/receive datatypes

form, however for `MPI_Alltoall` the data needs to be contiguous, which we achieve with `MPI_Type_create_resized`.

### 3.3.3 Hybrid OpenMP + MPI

The hybrid version of the algorithm is very close to the MPI version. We parallelized the update of the inner grid cells as we did in the pure OpenMP version. We also added a `#pragma omp parallel for` clause to the local transposing of the blocks (if applicable) and the reaction part of the simulation.

```

1 // build contiguous (rows) vectors for boundaries
2 MPI_Type_contiguous(Ny_loc, MPI_DOUBLE, &bottom_boundary);
3 MPI_Type_commit(&bottom_boundary);
4
5 MPI_Type_contiguous(Ny_loc, MPI_DOUBLE, &top_boundary);
6 MPI_Type_commit(&top_boundary);
```

Listing 9: MPI data types for ghost cells

```

1 MPI_Datatype block_send, block_col, block_recv;
2
3
4 // send data type
5 int sizes[2] = {Nx_loc, Ny_loc}; // size of global array
6 int subsizes[2] = {Nx_loc, Nx_loc}; // size of sub-region
7 int starts[2] = {0,0}; // where does the first subarray begin
8
9 MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,←
10   MPI_DOUBLE, &block_send);
11 MPI_Type_commit(&block_send);
12
13 // resize -> make contiguous
14 MPI_Type_create_resized(block_send, 0, Nx_loc*sizeof(double), &←
15   block_resized_send);
16 MPI_Type_free(&block_send);
17 MPI_Type_commit(&block_resized_send);
18
19 // receive data type (only used when not transposing locally)
20 // create one row of the block
21 MPI_Type_vector(Nx_loc, 1, Ny_loc, MPI_DOUBLE, &block_col);
22 MPI_Type_commit(&block_col);
23 // combine rows into block
24 MPI_Type_hvector(Nx_loc, 1, sizeof(double), block_col, &←
25   block_recv);
26 MPI_Type_free(&block_col);
27 MPI_Type_commit(&block_recv);
28
29 // resize -> make contiguous
30 MPI_Type_create_resized(block_recv, 0, sizeof(double), &←
31   block_resized_recv);
32 MPI_Type_free(&block_recv);
33 MPI_Type_commit(&block_resized_recv);

```

Listing 10: MPI data types for transposing the grid

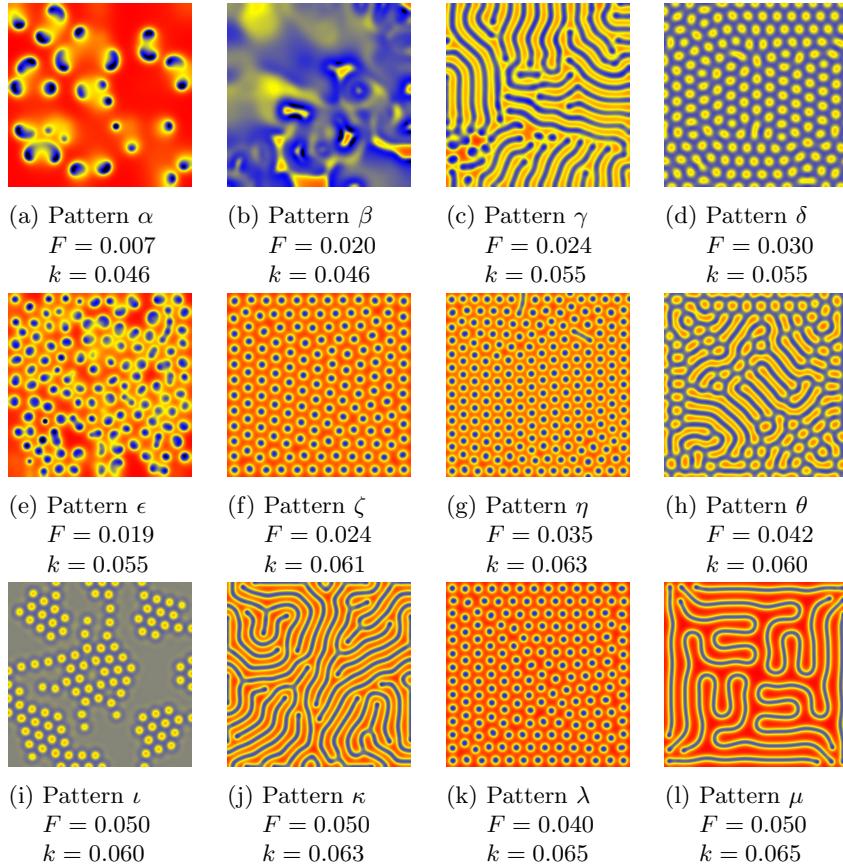


Figure 3: Patterns after  $10^5$  steps

## 4 Results

### 4.1 Patterns

We compared our resulting patterns with the patterns Pearson showed in his paper [2]. Most patterns are close to the ones Pearson got, the pattern  $\iota$  differs most. However, these differences are likely due to having slightly different parameters than Pearson, as we had to read them out of the parameter space.

### 4.2 Serial version

For the serial version, we ran it with and without compiler vectorization. With vectorization, we compiled using `g++ -ftree-vectorize ...`, without we used `g++ -fno-tree-vectorize ...`. The comparison of the system size scaling of these two version can be seen in figure 4. It is clearly visible, that it doesn't make a difference whether the compiler vectorizes the code or not. We fitted a line with python `numpy.polyfit` and obtained a scaling exponent of 2.2.

### 4.3 OpenMP version

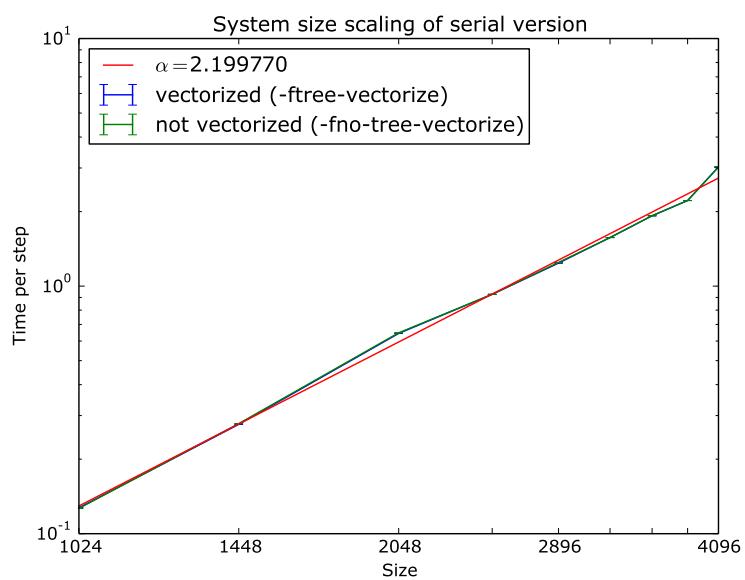


Figure 4: System size scaling of the serial version with and without compiler vectorization

## References

- [1] I.J.D. Craig and A.D. Sneyd. An alternating-direction implicit scheme for parabolic equations with mixed derivatives. *Computers & Mathematics with Applications*, 16(4):341350, 1988.
- [2] J E Pearson. Complex patterns in a simple system. *Science (New York, N.Y.)*, 261(5118):189192, 1993.
- [3] Wikipedia. Tridiagonal matrix algorithm — Wikipedia, the free encyclopedia, 2014. [Online; Version from 9 April 2014].