ADI for Reaction-Diffusion Systems

Stephanie Christ

August 4, 2015

1 Gray-Scott Reaction-Diffusion System

Reaction-Diffusion systems describe how the concentrations of chemical substances change 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:

$$U + 2V \to 3V$$
$$V \to P$$

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

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

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

The model is in 2 dimensions.

2 Numerical Method

We discretize the two-dimensional domain regularly in both dimensions, so that we obtain a square grid on which we perform the computation.

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. The diffusion of u and v can both be computed separately in each time step, as their contributions to the other equation is only explicitly in the reaction term.

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]. It is a finite different method which is unconditionally stable and second order in both time and space. The idea is to split each time step into two half-steps, one of which is implicit in x-direction and explicit in y-direction, for the other, the roles are switched. This leads us to tridiagonal matrix systems that are solvable in $\mathcal{O}(n)$. The algorithm is as follows:

First half-step:

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

Second half-step:

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

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

$$\frac{\partial \rho_{i,j}}{\partial x^2} = \frac{\rho_{i-1,j} - 2\rho_{i,j} + \rho_{i+1,j}}{\Delta x^2} \tag{5}$$
$$\frac{\partial \rho_{i,j}}{\partial y^2} = \frac{\rho_{i,j-1} - 2\rho_{i,j} + \rho_{i,j+1}}{\Delta y^2} \tag{6}$$

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

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

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

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

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

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

The second half-step gives a tridiagonal system to solve for each column of the domain. The tridiagonal matrix of this step is identical to the one from the first half step, as our grid is square with $\Delta y = \Delta x$.

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. The system wa want to solve is the following:

$$Ax = d \tag{10}$$

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

The method first modifies the coefficients in the following way:

$$c_i' = \begin{cases} \dfrac{c_i}{b_i} & ; \quad i = 0 \\ \dfrac{c_i}{b_i - a_i c_{i-1}'} & ; \quad i = 1, 2, \ldots, n-1 \end{cases} \tag{12}$$

$$d_i' = \begin{cases} \dfrac{d_i}{b_i} & ; \quad i = 0 \\ \dfrac{d_i - a_i d_{i-1}'}{b_i - a_i c_{i-1}'} & ; \quad i = 1, 2 \ldots, n. \end{cases} \tag{13}$$

Then the solution is obtained by backwards substitution.

$$x_n = d_n' \tag{14}$$

$$x_i = d_i' - c_i' x_{i+1} \qquad ; \; i = n-1, n-2, \ldots, 0. \tag{15}$$

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.

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

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

2.3 Boundary conditions

We use reflecting boundary conditions. This modifies the tridiagonal matrix system such that the values on each diagonal are not a constant. The first value on the upper diagonal c_0 and the last value on the lower diagonal a_n are doubled. This stems from the fact that the term outside of the domain is reflected inwards, e.g. a_0 is reflected onto and thus added to c_0. Since the lower and upper diagonal in this problem contain the same values, $a_0 + c_0 = 2c_0$. The same reasoning holds for a_n.

Our final matrix for both the first and the second half step looks the following way:

$$\begin{bmatrix} b & 2c & & & 0 \\ a & b & c & & \\ & a & b & \ddots & \\ & & \ddots & \ddots & c \\ 0 & & & 2a & b \end{bmatrix} \tag{18}$$

with

$$a = -\frac{D_\rho \Delta t}{2\Delta x^2} \qquad b = 1 + \frac{D_\rho \Delta t}{\Delta x^2} \qquad c = -\frac{D_\rho \Delta t}{2\Delta x^2} \tag{19}$$

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 where we store the three diagonals also each in a `std::vector<double>`. The class is mainly to simplify the access to the diagonals and to be able to pass the matrix as one object to the solver.

To visualize the result, we used OpenGL for displaying the changing concentration of u during the computation and LodePNG[1] – 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 listing 1). To facilitate the access to a grid cell, we have defined macros that calculate the index of the cell in the vector from the two-dimensional coordinates of the cell. 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 `i` and `j` switched as loop variables.

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

3.2 Thomas Algorithm

For the implementation of the Thomas Algorithm, we used the version on wikipedia from April 2014[2] We modified it so that we can pass the whole grid as an argument (see listing 3 for the declaration of the function). Since the access to the grid is to only one row/column for each tridiagonal system we need to solve, 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 has different values 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 both loops together with
`#pragma omp parallel for collapse(2)`.

[1] `http://lodev.org/lodepng`

[2] `https://en.wikipedia.org/w/index.php?title=Tridiagonal_matrix_algorithm&oldid=603501573`

```
// loop over all rows

// j=0
for (int i=0; i<N_; ++i) {
    uRhs[i] = U(i,0) + uCoeff * (U(i,1) - U(i,0));
}
TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,0), 1);

// inner grid points
for (int j=1; j<N_-1; ++j) {
    // create right-hand side of the systems
    for (int i=0; i<N_; ++i) {
        uRhs[i] = U(i,j) + uCoeff * (U(i,j+1) - 2.*U(i,j) + U(i, j-1));
    }

    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,j), 1);
}

// j=N_-1
for (int i=0; i<N_; ++i) {
    uRhs[i] = U(i,N_-1) + uCoeff * (- U(i,N_-1) + U(i,N_-2));
}
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)

```
double tmp;
for (int i=0; i<N_; ++i) {
    for (int j=0; j<N_; ++j) {
        tmp = U(i,j);
        U(i,j) += dt_ * ( -tmp*V(i,j)*V(i,j) + F_*(1.-tmp) );
        V(i,j) += dt_ * ( tmp*V(i,j)*V(i,j) - (F_+k_)*V(i,j) );
    }
}
```

Listing 2: Reaction, serial implementaion

```
namespace TriDiagMatrixSolver
{
    /**
     * Solve a tridiagonal matrix system.
     *
     * @param n          number of elements in the result
     * @param mat        tridiagonal matrix
     * @param rhs        right-hand side of the system
     * @param result     vector for the result, pointer to the first element
     * @param inc        increment for the elements of the result
     */
    void solve(unsigned int n, const TriDiagMatrix& mat, const std::vector<double>& rhs, ↩
        double *result, const unsigned int inc);
};
```

Listing 3: Declaration of our implementation of the Thomas algorithm

```
#pragma omp parallel num_threads(nthreads_)
{ // PARALLEL REGION BEGIN

// right hand sides for u and for v
std::vector<double> puRhs(N_);
std::vector<double> pvRhs(N_);

#pragma omp for
for (int j=1; j<N_-1; ++j) {
    // row-wise computations, see listing 1
}

} // PARALLEL REGION END
```

Listing 4: Parallel OpenMP region

```
MPI_Request request[8];
MPI_Status status[8];

// exchange boundaries
if (world.coord_x % 2 == 0) {

    MPI_Isend(&u_[(Nx_loc)*(Ny_loc)],   1, bottom_boundary, world.bottom_proc, TAG, cart_comm,↩
        &request[0]);
    MPI_Irecv(&u_[(Nx_loc+1)*(Ny_loc)], 1, bottom_boundary, world.bottom_proc, TAG, cart_comm,↩
        &request[1]);
    MPI_Isend(&u_[(Ny_loc)],            1, top_boundary,    world.top_proc,    TAG, cart_comm,↩
        &request[2]);
    MPI_Irecv(&u_[0],                   1, top_boundary,    world.top_proc,    TAG, cart_comm,↩
        &request[3]);
}
else {

    MPI_Irecv(&u_[0],                   1, top_boundary,    world.top_proc,    TAG, cart_comm,↩
        &request[0]);
    MPI_Isend(&u_[(Ny_loc)],            1, top_boundary,    world.top_proc,    TAG, cart_comm,↩
        &request[1]);
    MPI_Irecv(&u_[(Nx_loc+1)*(Ny_loc)], 1, bottom_boundary, world.bottom_proc, TAG, cart_comm,↩
        &request[2]);
    MPI_Isend(&u_[(Nx_loc)*(Ny_loc)],   1, bottom_boundary, world.bottom_proc, TAG, cart_comm,↩
        &request[3]);

}
```

Listing 5: Communication of the ghost cells

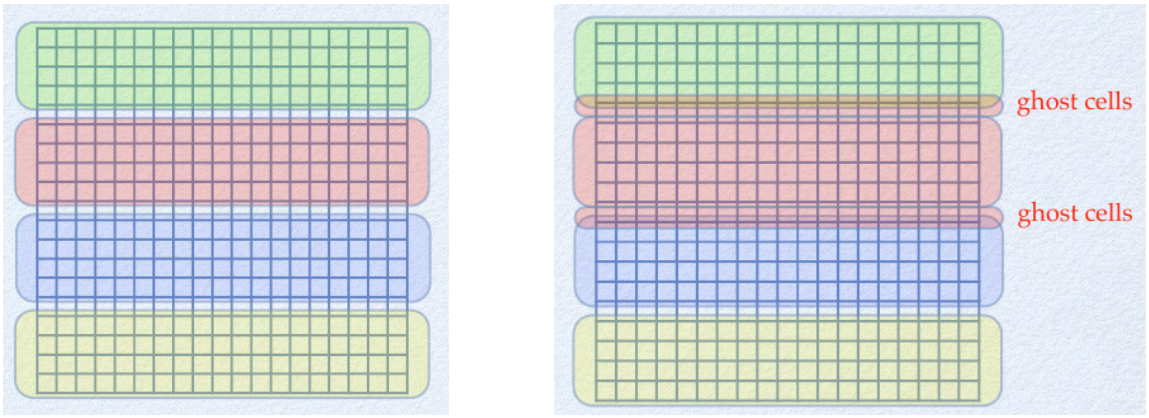
3.3.2 MPI

For the MPI version, we decompose the domain and distribute it between the processes.

First, the diffusion equation is solved. A half-step is executed in the following way: First, the boundary cells are communicated between the processes. Then, each processor solves the diffusion for its part of the grid. Lastly, the grid is transposed. After transposing the grid, the second half-step is identical to the first half-step. The second step transposes the grid back after the computations.

The reaction part of the simulation isn't changed from the serial implementation, except that the double loop in lines 2-3 in listing 2 is only over the local grid, such that each process computes the reaction for its section of the grid.

Domain decomposition 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. See listing 9 in section5.1 for the implementation of the first half-step.



(a) Domain decomposition (b) Ghost cells (for red)

Figure 1: Domain decomposition for 4 MPI processes[3]

```
MPI_Alltoall(&uTemp[Ny_loc], 1, block_resized_send, &u_[Ny_loc], 1, block_resized_send, ↩
    MPI_COMM_WORLD);

// locally transpose blocks
int ind1, ind2;
// loop over blocks
for (int b=0; b<Nb_loc; ++b) {
    for (int i=0; i<Nx_loc; ++i) {
        for (int j=0; j<i; ++j) {
            ind1 = (i+1)*Ny_loc + j + b*Nx_loc; // regular index + offset of block
            ind2 = (j+1)*Ny_loc + i + b*Nx_loc; // switch i and j

            std::swap(u_[ind1], u_[ind2]);
        }
    }
}
```

Listing 6: Transposing the grid using MPI Alltoall and local transpose

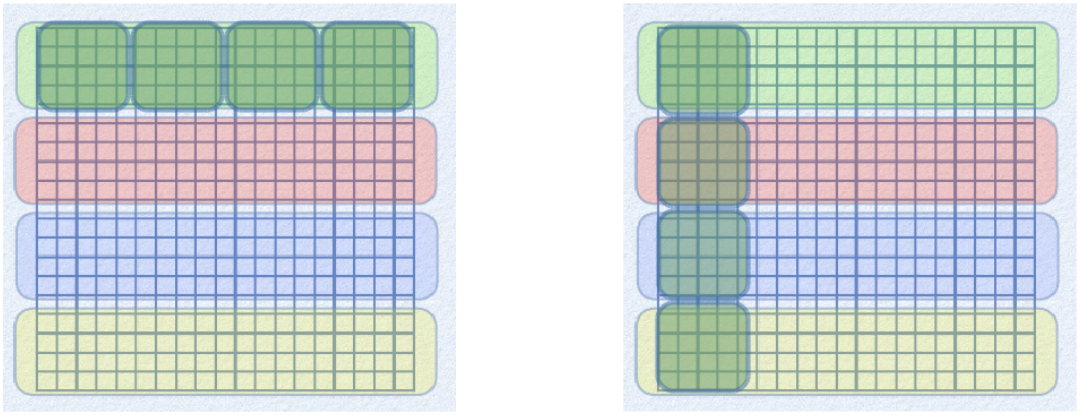
Communication The ghost cells are sent/received by the processes as shown in listing 5. We used nonblocking communication, because that lets us update the local inner grid cells (independent of ghost cells), while we wait on the ghost cells to arrive. After updating the inner grid cells, we need to wait 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 6). The second version uses a different receive data type than send data type in `MPI_Alltoall`. The receiving data type is ordered in a way that is the transpose of the send data type (see listing 7 in section 5.2). This lets us skip the local transposing of the blocks, as the received data is already stored transposed. We compare the performance of these two versions in section 4.4.

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

Listing 7: Transposing the grid using MPI Alltoall and different send/receive datatypes

[3]Figures from HPCSE slides



(a) Row of blocks before transposing (for green) (b) Row (now column) of blocks after transposing

Figure 2: Domain decomposition for MPI transpose[4]

```
// build contiguous (rows) vectors for boundaries
MPI_Type_contiguous(Ny_loc, MPI_DOUBLE, &bottom_boundary);
MPI_Type_commit(&bottom_boundary);

MPI_Type_contiguous(Ny_loc, MPI_DOUBLE, &top_boundary);
MPI_Type_commit(&top_boundary);
```

Listing 8: MPI data types for ghost cells

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 8). 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 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.

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 ι 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.

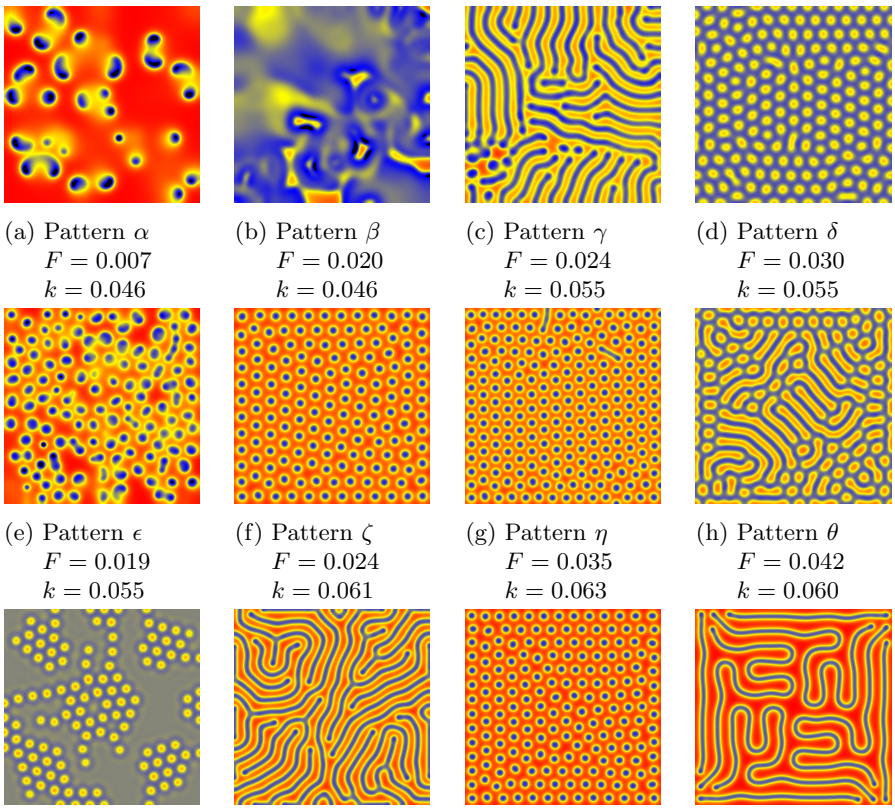
4.2 Serial version

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's `numpy.polyfit` and obtained a scaling exponent of 2.2.

4.3 OpenMP version

If we look at the strong scaling plot for the pure OpenMP implementation (figure 5, speedup with respect to the same implementation run with one thread), we can see that – depending on the system size – the algorithm scales

[4]Figures adapted from HPCSE slides



(a) Pattern α
$F = 0.007$
$k = 0.046$

(b) Pattern β
$F = 0.020$
$k = 0.046$

(c) Pattern γ
$F = 0.024$
$k = 0.055$

(d) Pattern δ
$F = 0.030$
$k = 0.055$

(e) Pattern ϵ
$F = 0.019$
$k = 0.055$

(f) Pattern ζ
$F = 0.024$
$k = 0.061$

(g) Pattern η
$F = 0.035$
$k = 0.063$

(h) Pattern θ
$F = 0.042$
$k = 0.060$

(i) Pattern ι
$F = 0.050$
$k = 0.060$

(j) Pattern κ
$F = 0.050$
$k = 0.063$

(k) Pattern λ
$F = 0.040$
$k = 0.065$

(l) Pattern μ
$F = 0.050$
$k = 0.065$

Figure 3: Patterns after 10^5 steps

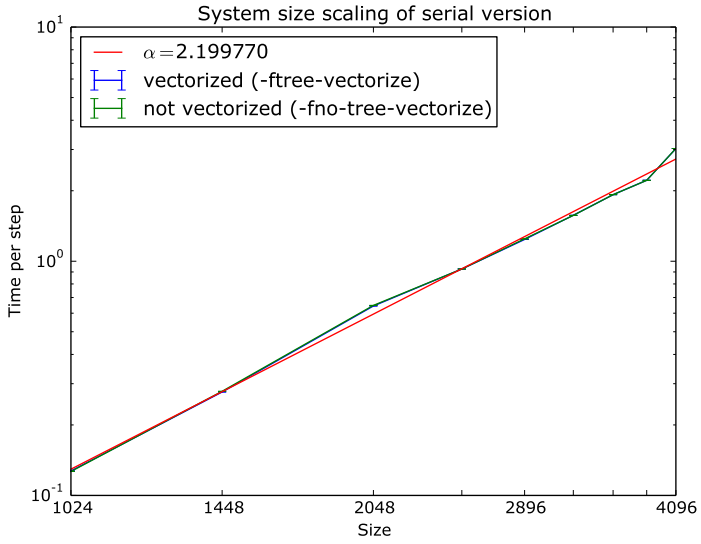


Figure 4: System size scaling of the serial version with and without compiler vectorization

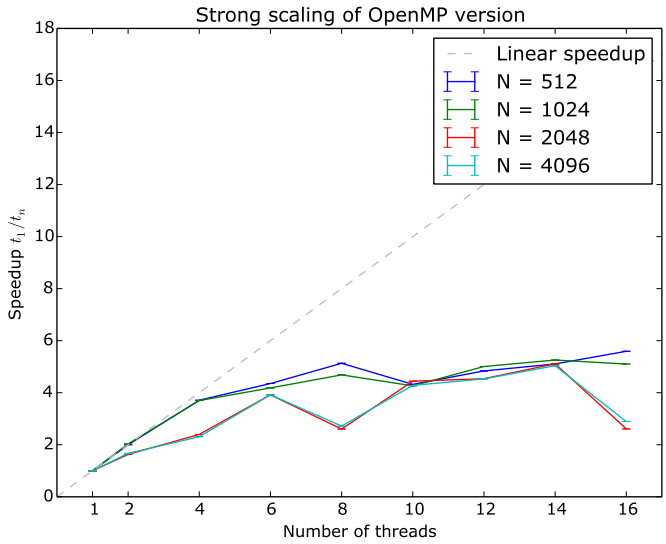


Figure 5: Strong scaling of the OpenMP version

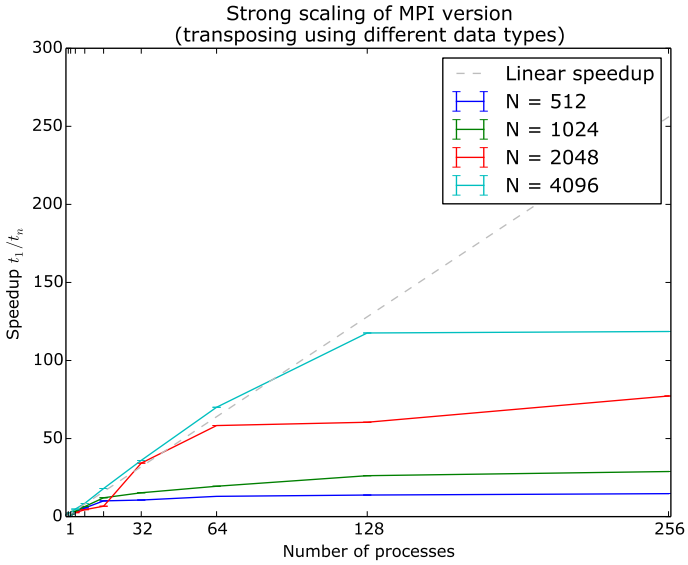


Figure 6: Strong scaling of the MPI version, using different data types for transposing the grid

up to 6 or 8 threads. The reason for that is that we only have available 8 physical cores, after that we enabled hyperthreading to go up to 16 threads. It is clear that hyperthreading isn't of use here. Also for the up to 6 or 8 threads, it doesn't scale well. This might be due to the fact that we allocate the grids NUMA (Non-Uniform Memory Access)-aware. To implement the OpenMP NUMA-aware, each thread would need to initialize its part of the memory, as opposed to having one thread initializing the whole memory. To be able to do that, the thread which touches the memory first has to decide where in the memory data gets placed. This is accomplished by using a custom allocator when one thread allocates the memory before having all threads initialize their part.

4.4 MPI version

The scaling plot for pure MPI parallelization can be seen in figure 6. Speedup has been calculated for different system sizes with respect to the runtime of the MPI implementation run with one process. We can see that the larger the system is, the longer it scales. This is expected, as with more processes, the communication overhead gets larger and quickly overpowers the time saved by parallelism in small systems. If we, for example, work on a 512×512 grid with 256 processes, each task will only work on a $2 \times 512 = 2 \times N$ grid. In this example, `MPI_Alltoall` needs to send and receive $128 \cdot 128$ blocks of size 2×2.

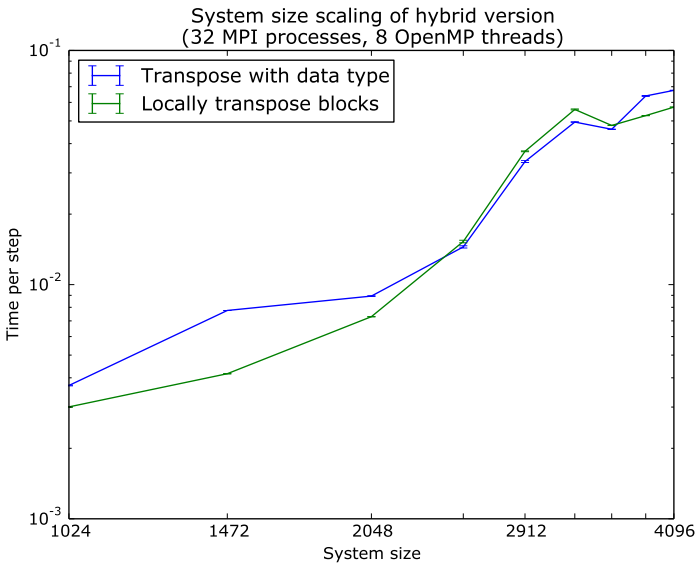


Figure 7: System size scaling scaling of the hybrid version, $N = 4096$, with 8 OpenMP threads and 32 MPI processes

4.5 Hybrid version

For the hybrid version, we can scaling with respect to system size, OpenMP threads and MPI processes.

4.5.1 System size scaling

We compare the size scaling of the hybrid version for the two different ways to transpose the grid. The simulation has been run with 8 OpenMP threads and 32 MPI processes. As can be seen in figure 7, both version scale similarly. For the version which locally transposes the blocks, we got a scaling exponent of 2.4, for the other version with the different data types, we got 2.2.

4.5.2 OpenMP scaling

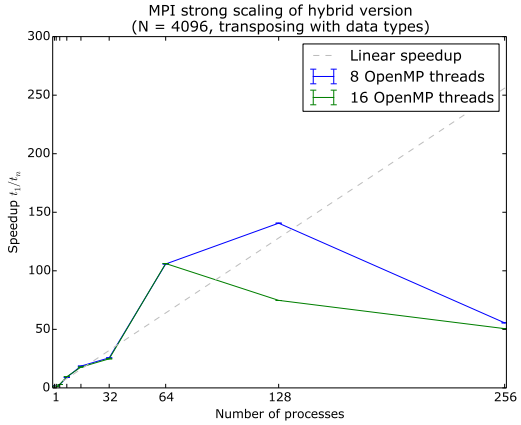
For the OpenMP scaling, we ran the hybrid version with 32 MPI processes. As the OpenMP version doesn't scale well with the number of threads, neither does the hybrid version (see figure 9). The reasoning is the same as for the OpenMP version.

4.5.3 MPI scaling

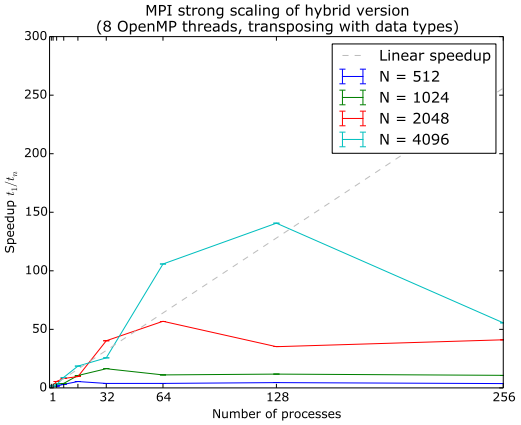
The scaling plot for strong scaling with respect to the number of MPI processes can be seen in figure 8. Here, the grid has been transposed using different data types. The speedup is calculated with respect to the hybrid implementation ran with one MPI process and the same number of OpenMP threads as the other data points (8 or 16).

In figure 8a we can see that up to 64 processes, the versions with 8 and with 16 threads scale exactly the same. This shows that hyperthreading (16 threads) doesn't make sense on this architecture. With 8 threads, it scales up to 128 MPI processes, with 16, it doesn't scale anymore after 64 processes. The reason for this is that at 128 processes, each process has to solve the ADI-steps for only 32 rows, and splitting such a small number between 16 threads doesn't make it any faster, since the overhead is too big.

Figure 8b compares the scaling for different system sizes. We can see that the larger the system is, the better it scales. The reasoning is the same as for the MPI version.



(a) Comparison of strong scaling with 8 and 16 OpenMP threads, $N = 4086$



(b) Comparison of strong scaling of different system sizes, with 8 OpenMP threads

Figure 8: MPI strong scaling of the hybrid version, using different data types for transposing the grid

5 Appendix

5.1 First half-step using MPI

```
// send/receive ghost cells (non-blocking)

// update inner grid points (see listing 1)

// wait for boundaries to arrive
MPI_Waitall(8,request,status);


// update local boundaries

if (world.rank == 0) {
    // i=0 local and global
    for (int j=0; j<N_; ++j) {
        uRhs[j] = U(0,j) + uCoeff * (U(1,j) - U(0,j));
    }
    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(0,0), 1);
}
else {
    // i=0 local
    for (int j=0; j<N_; ++j) {
        uRhs[j] = U(0,j) + uCoeff * (U(0+1,j) - 2.*U(0,j) + U(0-1,j));
    }
    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(0,0), 1);
}


if (world.rank == world.size-1) {
    // i=Nx_loc-1 local and i=N_-1 global
    for (int j=0; j<N_; ++j) {
        uRhs[j] = U(Nx_loc-1,j) + uCoeff * (- U(Nx_loc-1,j) + U(Nx_loc-2,j));
    }
    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(Nx_loc-1,0), 1);
}
else {
    // i=Nx_loc-1 local
    for (int j=0; j<N_; ++j) {
        uRhs[j] = U(Nx_loc-1,j) + uCoeff * (U(Nx_loc-1+1,j) - 2.*U(Nx_loc-1,j) + U(Nx_loc-1-1,↩
            j));
    }
    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UTEMP(Nx_loc-1,0), 1);
}
```

Listing 9: First half-step parallelized with MPI

5.2 Datatypes for MPI transpose

```
MPI_Datatype block_send, block_col, block_recv;


// send data type
int sizes[2]    = {Nx_loc, Ny_loc}; // size of global array
int subsizes[2] = {Nx_loc, Nx_loc}; // size of sub-region
int starts[2]   = {0,0}; // where does the first subarray begin

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C, MPI_DOUBLE, &block_send);
MPI_Type_commit(&block_send);

// resize -> make contiguous
MPI_Type_create_resized(block_send, 0, Nx_loc*sizeof(double), &block_resized_send);
MPI_Type_free(&block_send);
MPI_Type_commit(&block_resized_send);


// receive data type (only used when not transposing locally)
// create one row of the block
MPI_Type_vector(Nx_loc, 1, Ny_loc, MPI_DOUBLE, &block_col);
MPI_Type_commit(&block_col);
```

```
// combine rows into block
MPI_Type_hvector(Nx_loc, 1, sizeof(double), block_col, &block_recv);
MPI_Type_free(&block_col);
MPI_Type_commit(&block_recv);

// resize -> make contiguous
MPI_Type_create_resized(block_recv, 0, sizeof(double), &block_resized_recv);
MPI_Type_free(&block_recv);
MPI_Type_commit(&block_resized_recv);
```

Listing 10: MPI data types for transposing the grid

5.3 OpenMP strong scaling of hybrid version

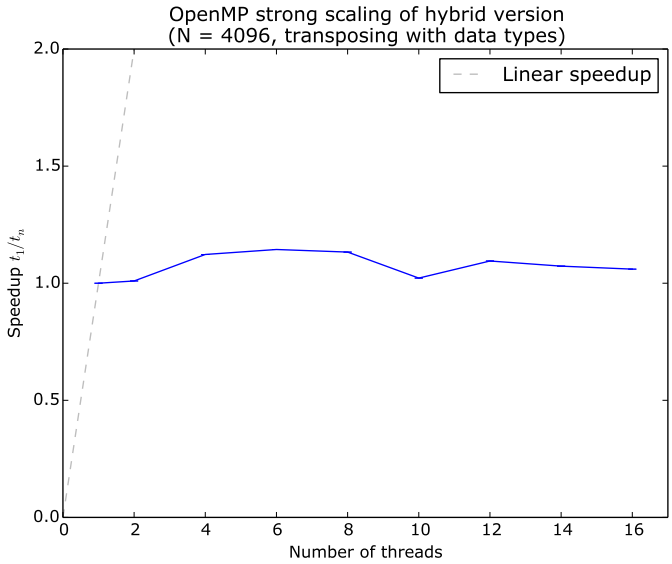


Figure 9: OpenMP strong scaling of the Hybrid version, $N = 4096$, 32 MPI processes, using different data types for transposing the grid

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.