ADI for Reaction-Diffusion Systems

Lukas Strebel
Computational Sciences & Engineering BSc
ETH Zürich, Switzerland
August 3, 2015

This report is the conclusion to the project of the course High Performance Computing for Science and Engineering taught by P. Koumoutsakos and M. Troyer in the spring semester 2015.

1 Introduction

2 Gray-Scott Reaction-Diffusion - Model description

The Gray-Scott reaction-diffusion system [1] combines the interaction of two chemical substances and their diffusion into one set of differential equations. The reaction between the substances U and V is described by two simple irreversible reactions:

$$\begin{aligned} U + 2V &\rightarrow 3V \\ V &\rightarrow P \end{aligned} \tag{1}$$

where P is an inert product.

2.1 Governing Equations

Reaction (1) and diffusion in dimensionless units combined result in the following equations:

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

where k is the dimensionless rate constant of the second reaction and F is the dimensionless feed rate constant. D_u and D_v are the diffusion coefficients.

3 Discretized system

The model is applied to a 2-dimensional domain which is discretized into a square grid with uniform, square grid cells.

To solve equations (2) the reaction and diffusion are treated separately. Diffusion is handled first and the reaction is added afterwards to the solution of the diffusion. Since u and v are only coupled in the reaction their diffusion can be handled separately.

3.1 Diffusion - Alternating Direction Implicit (ADI) method

The general diffusion equation for a substance ρ (in the Gray-Scott model u or v) is

$$\frac{\partial \rho}{\partial t} = D_\rho \Delta \rho \tag{3}$$

This equation could be solved by explicit or implicit methods. However the problem with explicit methods like forward euler is instability i.e. the numerical solution will diverge from the exact solution exponentially over time. To remain stable forward euler has a strict condition on the time step:

$$\Delta t < \frac{\Delta x^2 \Delta y^2}{D_\rho(\Delta x^2 + \Delta y^2)} \tag{4}$$

which means for square grid cells that $\Delta t = \Theta(h^2)$.

Implicit methods on the other hand are unconditionally stable but require a system of linear equations to be solved each time step.

The idea of the Alternating Direction Implicit (ADI) method [2] is to combine explicit and implicit methods in a smart way to achieve an unconditionally scheme that is second order in space and time.

The ADI method does this by splitting each time step into two half-steps and then treating the x-direction implicitly and the y-direction explicitly in one half-step and with switched directions in the other half-step. This way each half-step consists of solving m tridiagonal systems of equations, where m is the size of the grid in the explicitly treated direction. The scheme looks like this:

First half-step:

$$\rho_{i,j}^{n+\frac{1}{2}} = \rho_{i,j}^{n} + \frac{D_\rho \delta t}{2}\left[\frac{\partial^2 \rho_{i,j}^{n+\frac{1}{2}}}{\partial x^2} + \frac{\partial^2 \rho_{i,j}^{n}}{\partial y^2}\right] \tag{5}$$

Second half-step:

$$\rho_{i,j}^{n+1} = \rho_{i,j}^{n+\frac{1}{2}} + \frac{D_\rho \delta t}{2}\left[\frac{\partial^2 \rho_{i,j}^{n+\frac{1}{2}}}{\partial x^2} + \frac{\partial^2 \rho_{i,j}^{n+1}}{\partial y^2}\right] \tag{6}$$

Where i is the index for the grid in x-direction and j is the index for the grid in y-direction. The superscript n denotes the time step.

The second derivative is approximated using a second order central differences scheme.

$$\begin{aligned} \frac{\partial \rho_{i,j}}{\partial x^2} &= \frac{\rho_{i-1,j} - 2\rho_{i,j} + \rho_{i+1,j}}{\Delta x^2} \\ \frac{\partial \rho_{i,j}}{\partial y^2} &= \frac{\rho_{i,j-1} - 2\rho_{i,j} + \rho_{i,j+1}}{\Delta y^2} \end{aligned} \tag{7}$$

For square grid cells $\Delta x = \Delta y$. Inserting the central differences (7) in the first half step equation (5) results in:

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

Denoting the constant coefficients as $C = \frac{D_\rho \delta t}{2\Delta x^2}$ and moving the implicit parts to the right hand side results in:

$$-C\rho_{i-1,j}^{n+\frac{1}{2}} + (1+2C)\rho_{i,j}^{n+\frac{1}{2}} - C\rho_{i+1,j}^{n+\frac{1}{2}} = C\rho_{i,j-1}^{n} + (1-2C)\rho_{i,j}^{n} - C\rho_{i,j+1}^{n} \tag{9}$$

This equation describes a tridiagonal system with the constants $-C$ on the lower and upper diagonal and the constant $(1+2C)$ on the middle diagonal. This tridiagonal system has to be solved for each row of the domain grid during the first half-step.

Analogously the following equation for the second half-step (6) can be derived:

$$-C\rho_{i,j-1}^{n+1} + (1+2C)\rho_{i,j}^{n+1} - C\rho_{i,j+1}^{n+1} = C\rho_{i-1,j}^{n+\frac{1}{2}} + (1-2C)\rho_{i,j}^{n+\frac{1}{2}} - C\rho_{i+1,j}^{n+\frac{1}{2}} \tag{10}$$

Which describes the same tridiagonal system to solve for each column of the domain grid during the second half-step.

3.1.1 Thomas Algorithm

One of the advantages of ADI is that it reduces the implicit part of the calculations into tridiagonal systems as described above. This means instead of general linear system solvers a more efficient algorithm for tridiagonal systems can be used.

The Thomas algorithm is such a specialized algorithm to solve tridiagonal systems in $\mathcal{O}(n)$.

A general tridiagonal system for $n+1$ unknowns may be written as

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \tag{11}$$

with $a_0 = c_n = 0$.

The Thomas algorithm consists of two parts, in the first part a forward sweep modifies the coefficients as follows:

$$\begin{aligned} c'_i &= \begin{cases} \dfrac{c_i}{b_i} & ; \quad i = 0 \\ \dfrac{c_i}{b_i - a_i c'_{i-1}} & ; \quad i = 1, 2, \ldots, n-1 \end{cases} \\ d'_i &= \begin{cases} \dfrac{d_i}{b_i} & ; \quad i = 0 \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & ; \quad i = 1, 2 \ldots, n. \end{cases} \end{aligned} \tag{12}$$

Where primes denote the modified coefficients. In the second part the solution is gathered by back substition:

$$\begin{aligned} x_n &= d'_n \\ x_i &= d'_i - c'_i x_{i+1} \qquad ; \; i = n-1, n-2, \ldots, 0. \end{aligned} \tag{13}$$

To note is that the algorithm's forward and backward part are basically for-loops with dependencies between iterations and therefore not easily parallelizable.

3.2 Reaction - Forward Euler scheme

For the reaction term of the model equations (2) a forward Euler scheme can be applied. As noted before the reaction has to be done after the diffusion and include the solution of the diffusion equation (denoted by $\tilde{u}$ and $\tilde{v}$)

$$\begin{aligned} u_{i,j}^{n+1} &= \tilde{u}_{i,j}^{n+1} + \Delta t \left(-\tilde{u}_{i,j}^{n+1} (\tilde{v}_{i,j}^{n+1})^2 + F(1 - \tilde{u}_{i,j}^{n+1}) \right) \\ v_{i,j}^{n+1} &= \tilde{v}_{i,j}^{n+1} + \Delta t \left(\tilde{u}_{i,j}^{n+1} (\tilde{v}_{i,j}^{n+1})^2 - (F+k) \tilde{v}_{i,j}^{n+1} \right) \end{aligned} \tag{14}$$

Noteworthy might be that the reaction terms (14) are only dependent on the values located at the corresponding grid point i.e. they are purely local operations and require no neighborhood interactions.

3.3 Initial conditions and constant coefficients

A simulation domain from $[-1, 1]^2$ was used in all implementations described in this report. To break up the square symmetry a square in the middle of the domain contains some random noise.

$$
\begin{aligned}
u(x,y,0) &= (1-\chi(x,y)) + \chi(x,y)\left(\frac{1}{2} + \frac{r_1}{100}\right) \\
v(x,y,0) &= \chi(x,y)\left(\frac{1}{4} + \frac{r_2}{100}\right) \\
\chi(x,y) &= \begin{cases} 1 & \text{for } (x,y) \in [-0.2, 0.2]^2 \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{15}
$$

where r_1 and r_2 are random numbers from a normal distribution $\mathcal{N}(0,1)$. In our implementation we used the Mersenne-Twister random number generator provided in the C++ standard library seeded with 42 to generate these numbers.

The diffusion coefficients were defined to be $D_u = 2 \cdot 10^{-5}$ and $D_v = 1 \cdot 10^{-5}$

The parameters F and k were defined based on the ones described in Complex Patterns in a Simple System by J.E. Pearson [3].

3.4 Boundary conditions

Reflecting boundary conditions are used in the implementations described here. This means that the boundaries have zero flux through their surface and everything gets reflected at them. In terms of the previously mentioned scheme this results in a small change in the tridiagonal systems (9) and (10). Specifically the first value on the upper diagonal c_0 and the last value on the lower diagonal a_n have to be doubled for both the first and second half-step.

4 Implementation

4.1 Serial

4.1.1 Outline

Our implementation is guided by the following basic outline:

- Initialize u and v according to the initial conditions (15).
- Then for each timestep do:
  - Start performance timer
  - Perform first half-step by :
    - Looping over all rows
    - Creating the right hand side
    - Solving the tridiagonal system with Thomas algorithm

  - Perform second half-step by:
    * Looping over all columns
    * Creating the right hand side
    * Solving the tridiagonal system with Thomas algorithm
  - Perform reaction on each grid point
  - Stop performance timer

- Calculate average time per step and standard deviation.

4.1.2 ADI half step implementation

The concrete implementation of a half-step:

```
/****************** DIFFUSION (ADI) ************************************/
    // perform the first half-step
    // 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));
        vRhs[i] = V(i,0) + vCoeff * (V(i,1) - V(i,0));
    }
    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,0), 1);
    TriDiagMatrixSolver::solve(N_, matV1_, vRhs, &VHALF(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));
            vRhs[i] = V(i,j) + vCoeff * (V(i,j+1) - 2.*V(i,j) + V(i,j-1));
        }

        TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,j), 1);
        TriDiagMatrixSolver::solve(N_, matV1_, vRhs, &VHALF(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));
        vRhs[i] = V(i,N_-1) + vCoeff * (- V(i,N_-1) + V(i,N_-2));
    }
    TriDiagMatrixSolver::solve(N_, matU1_, uRhs, &UHALF(0,N_-1), 1);
    TriDiagMatrixSolver::solve(N_, matV1_, vRhs, &VHALF(0,N_-1), 1);
```

Listing 1: Implementation of first half-step, second half-step is done analogously except the role of i and j are switched.

The values of u and v are stored row wise in a `std::vector<double>`. For ease of use preprocessor `#define` directives are used to access u and v via `U(i,j)` or `V(i,j)` respectively. The boundary rows in the first half-step and the boundary columns in the second half-step have to be done outside of the main loop because of their different dependencies on the neighbouring cells. `matU1` and `matV1` refer to the tridiagonal system described in the ADI section (9, 10). We encapsulated these tridiagonal system into their own class that stores the diagonals each in a `std::vector<double>` and provides access to them. Also that way they can be passed to the solver as a single object.

4.1.3 Thomas algorithm implementation

`TriDiagMatrixSolver::solve(...)` in listing 1 is our implementation of the Thomas algorithm, which is a modified version of the implementation found on Wikipedia in April 2014 [4].

```
1  void TriDiagMatrixSolver::solve(int n, const TriDiagMatrix& mat,
2  const std::vector<double>& rhs, double *result, unsigned int inc)
3  {
4
5      std::vector<double> a = mat.getL(); // lower diagonal, indexed 1..n-1
6      std::vector<double> b = mat.getM(); // main diagonal, indexed 0..n-1
7      std::vector<double> c = mat.getU(); // upper diagonal, indexed 0..n-2
8
9
10     n--; // since we start from x_0 (not x_1) (n now denotes the max index)
11     c[0] /= b[0];
12
13     std::vector<double> r(rhs);
14     r[0] /= b[0];
15
16     double tmp;
17
18     for (int i=1; i<n; ++i) {
19         tmp = b[i] - a[i]*c[i-1];
20         c[i] /= tmp;
21         r[i] = (r[i] - a[i]*r[i-1]) / tmp;
22     }
23
```

```
24      *(result + n*inc) = (r[n] - a[n]*r[n-1]) / (b[n] - a[n]*c[n-1]);
25
26      for (int i=n-1; i>=0; --i) {
27          *(result + i*inc) = r[i] - c[i] * (*(result + (i+1)*inc));
28      }
29  }
```

Listing 2: Implementation of Thomas algorithm.

In order to use the same function for each row/column and for both half-steps we use the `double *result` argument as the pointer to the first element of the row/column for which the function was called and the `unsigned int inc` argument to distinguish between row-wise access (for the first half-step `inc` = 1 and column-wise access for the second half-step `inc` = N.

4.1.4 Implementation of reaction

After the second half-step is done the reaction can be performed.

```
1       /****************** REACTION ***********************************************/
2
3       double tmp;
4       for (int i=0; i<N_; ++i) {
5           for (int j=0; j<N_; ++j) {
6               tmp = U(i,j);
7               U(i,j) += dt_ * ( -U(i,j)*V(i,j)*V(i,j) + F_*(1.-U(i,j)) );
8               V(i,j) += dt_ * ( tmp*V(i,j)*V(i,j) - (F_+k_)*V(i,j) );
9           }
10      }
```

Listing 3: Implementation of reaction.

As mentioned before the reaction equation (14) has to be done on each gridpoint locally and doesn't depend on any neighbours.

4.2 OpenMP implementation

Our parallelization with OpenMP consists of three parallel regions, one for each half-step to parallelize the loop over the inner grid points and one more to parallelize the reaction across the whole grid.

```
1   // inner grid points
2
3   #pragma omp parallel num_threads(nthreads_)
4       { // 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) {
        // create right-hand side of the systems
        for (int i=0; i<N_; ++i) {
            puRhs[i] = U(i,j) + uCoeff * (U(i,j+1) - 2.*U(i,j) + U(i,j-1));
            pvRhs[i] = V(i,j) + vCoeff * (V(i,j+1) - 2.*V(i,j) + V(i,j-1));
        }

        TriDiagMatrixSolver::solve(N_, matU1_, puRhs, &UHALF(0,j), 1);
        TriDiagMatrixSolver::solve(N_, matV1_, pvRhs, &VHALF(0,j), 1);
    }
    } // PARALLEL REGION END
```

Listing 4: Parallelization of the inner grid points in the first ADI half-step with OpenMP (second half-step done analogously).

As shown in listing 4 we parallelize the outer loop in the half-steps as they have only read-dependencies on the other rows (first half-step) / columns (second half-step). Each thread declares its own private right hand side inside the parallel region and writes the result of the tridiagonal solver in a region of the result vector that no other thread reads or writes to. This is due to the fact that the end of the parallel region after the first half-step represents a implicit barrier before the parallel region of the second half-step starts. All other variables are read-only and can therefore be safely shared by all threads.

```
#pragma omp parallel num_threads(nthreads_)
    { // PARALLEL REGION BEGIN
    double tmp;
#pragma omp for collapse(2)
    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) );
        }
    }
    } // PARALLEL REGION END
```

Listing 5: Parallelization of the reaction with OpenMP

Since the reaction as mentioned before is local to each grid point the parallelization of the two loops can be done together with the `collapse` option of the pragma omp for.

4.3 MPI implementation

For the parallelization with MPI on distributed memory the domain needs to be split equally among the processors. Because the first half step requires a processor to have full rows the simplest way to distribute the grid is block row-wise. This way each processor gets a set of neighbouring rows and can do its job on the local, inner rows without any communications with the other processors.

For the top and bottom row of each processor's local grid read access to one additional row of a neighbouring processor bottom or top row respectively is required. Since we work on a Cartesian grid we can use the MPI Cartesian topology to easily find the neighbours of a processors using the `MPI_Cart_shift` function. To do the local boundaries efficiently ghost rows are used and communicated using non-blocking methods so that the communication and the computation of the inner grid cells can be overlapped as shown in listing (6).

```
// 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]);
        // exactly the same for v, omitted here
    }
    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]);
        // exactly the same for v, omitted here
    }
```

```
24
25          // Do inner grid cells computations
```

Listing 6: Communication of ghost cells before each ADI half-step

After the local, inner grid cells are calculated the processors have to wait until the non-blocking communications are done and then the local boundaries can be computed. Special care has to be taken for the processors that are on the global grid boundaries which have to do the boundary calculations in the way described before (for example in listing (1)).
The thus far described MPI implementation works for the first half-step. The second half-step however requires each processor to have full columns and therefore we need to transpose the grid beforehand.

This means each processor needs to split its set of rows into blocks with the width of each block equal to the number of rows each processor has in its row block. These blocks then correspond to the grid points that have to be sent to the same processor. The blocks are sent using `MPI_Alltoall` but they still need to be transposed. To do the transpose we have two different approaches.

The first approach receives the blocks and transposes each block locally according to listing (7).

```
1           MPI_Alltoall(&uTemp[Ny_loc], 1, block_resized_send,
2   &u_[Ny_loc], 1, block_resized_send, MPI_COMM_WORLD);
3           MPI_Alltoall(&vTemp[Ny_loc], 1, block_resized_send,
4   &v_[Ny_loc], 1, block_resized_send, MPI_COMM_WORLD);
5
6           // locally transpose blocks
7           // loop over blocks
8           int ind1, ind2;
9           for (int b=0; b<Nb_loc; ++b) {
10              for (int i=0; i<Nx_loc; ++i) {
11                  for (int j=0; j<i; ++j) {
12                      ind1 = (i+1)*Ny_loc + j + b*Nx_loc;
13          // regular index + offset of block
14                      ind2 = (j+1)*Ny_loc + i + b*Nx_loc;
15          // switch i and j
16
17                      std::swap(u_[ind1], u_[ind2]);
18                      std::swap(v_[ind1], v_[ind2]);
19                  }
20              }
21          }
```

Listing 7: Approach one to transpose the grid based on `MPI_Alltoall` with one send/receive data type and a locally done transpose of each block.

The second approach changes the receive data type in such a way that is the transpose of the sent data type see listing (8).

```
    // build datatypes for transpose
    MPI_Datatype block_send, block_col, block_recv;

    // send data type

    // define the sizes for the blocks
    // size of the whole region the processor handles
    int sizes[2]    = {Nx_loc, Ny_loc};
    // size of sub-region (square) / blocks to be sent for transpose
    int subsizes[2] = {Nx_loc, Nx_loc};
    // where does the first subarray begin (which index)
    int starts[2]   = {0,0};

    // creating the blocks
    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 datatype ( for 2nd approach of transpose)

    // define a row of the transposed (receive) block
    // as vector with a stride of the length of the old row
    // this way each element of an old column will be in the same new row
    // and in the correct order
    MPI_Type_vector(Nx_loc, 1, Ny_loc, MPI_DOUBLE, &block_col);
    MPI_Type_commit(&block_col);

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

```
38      MPI_Type_commit(&block_recv);
39
40      // resize data structure, so that it is contiguous (for alltoall)
41      MPI_Type_create_resized(block_recv, 0, sizeof(double),
42   &block_resized_recv);
43      MPI_Type_free(&block_recv);
44      MPI_Type_commit(&block_resized_recv);
```

Listing 8: .

With a receive type defined like described the transpose after each half-step is done in a single call to `MPI_Alltoall` for u and one for v. (See listing (9))

```
1          // after half-step is done:
2
3          MPI_Alltoall(&uTemp[Ny_loc], 1, block_resized_send,
4  &u_[Ny_loc], 1, block_resized_recv, MPI_COMM_WORLD);
5          MPI_Alltoall(&vTemp[Ny_loc], 1, block_resized_send,
6  &v_[Ny_loc], 1, block_resized_recv, MPI_COMM_WORLD);
7
8          // ready for next half-step or reaction
```

Listing 9: .

After the second half-step including the transposing of the grid is done, each processor can perform the reaction on its own grid points.

4.4 Hybrid OpenMP + MPI implementation

Our hybrid version combines OpenMP and MPI by enhancing the pure MPI version with OpenMP threads to do the inner grid cells of each processor in parallel like the pure OpenMP version does it. OpenMP threads are also used to do reaction as well as the local transpose in parallel (if approach one is used and not approach two).

5 Results

5.1 Correctness

To see if our implementations produce the expected results we implemented a visualization using OpenGL to display the evolution of the concentration of u during the time stepping.

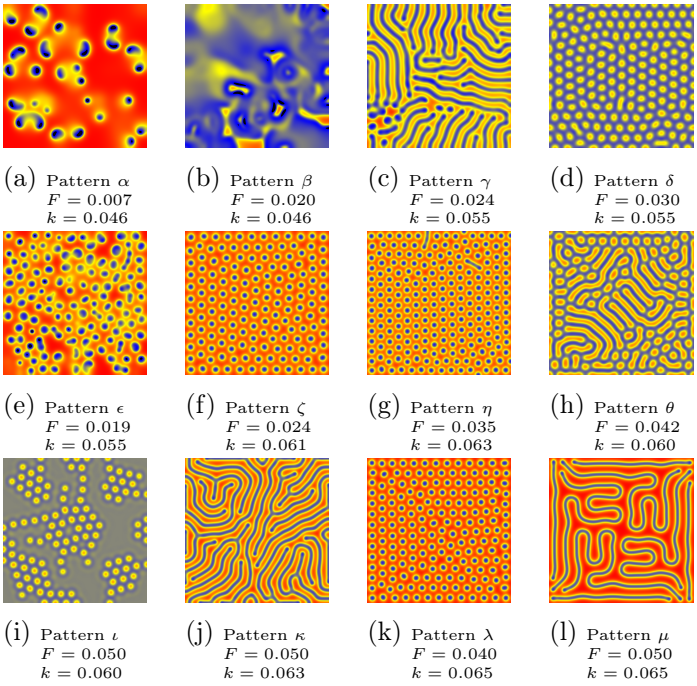

Figure 1: Concentration of u visualized after 10^5 steps

Since some patterns take a lot of time steps to develop we later used LodePNG [5] as a image encoder to save an image of the concentration of u at the end of simulation time only.

We tried to recreate the same patterns Pearson got in his paper [3], the results can be seen in Figure (1).

5.2 Performance

5.2.1 Test environment

To test the performance of our implementations we were allowed to use the Piz Daint supercomputer of the Swiss National Supercomputing Centre (CSCS). The Piz Daint is a Cray XC30 system and has 5272 compute nodes in total.

Each compute node features an Intel Xeon E5-2670 "Sandybridge" CPU running at a nominal frequency of 2.6 GHz with 8 physical cores per CPU, which can run 2 "hyperthread" hardware threads on each core and has 32 GB of memory.

Each node also features a Nvidia Tesla K20x "Kepler" GPU which sadly we didn't take advantage of in this project.

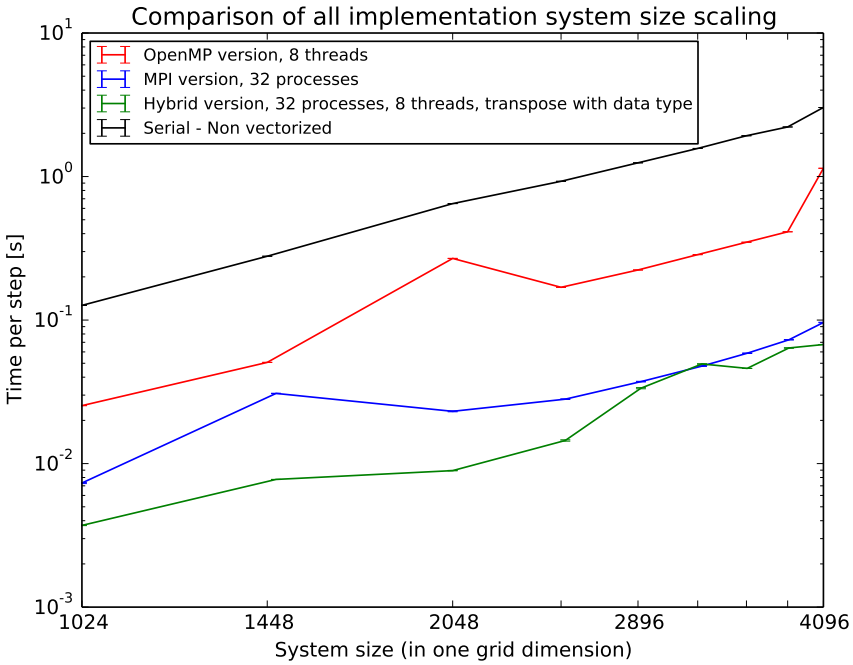

Figure 2: Comparison of the various implementations in regard to their scaling with system size

5.2.2 System size scaling comparison

In all the following measurements we took the time of each step in a simulation over 10^3 steps and averaged them. Also shown in each figure are the standard deviations of the measurements as error bars (which in all cases are barely visible).

In figure (2) we compared how the different implementations scale with the system size. For the serial implementation we took these measurements for a non vectorized as well as a version compiled with `g++ -ftree-vectorize` but we observed no significant difference between the two and therefore the vectorized version was omitted.

Not suprisingly the hybrid MPI + OpenMP version shows the best overall scaling with system size. However for very large systems the hybrid version approaches the pure MPI version, which could be seen as a hint that the communication between nodes and not how fast each node does its computation becomes the limiting factor.

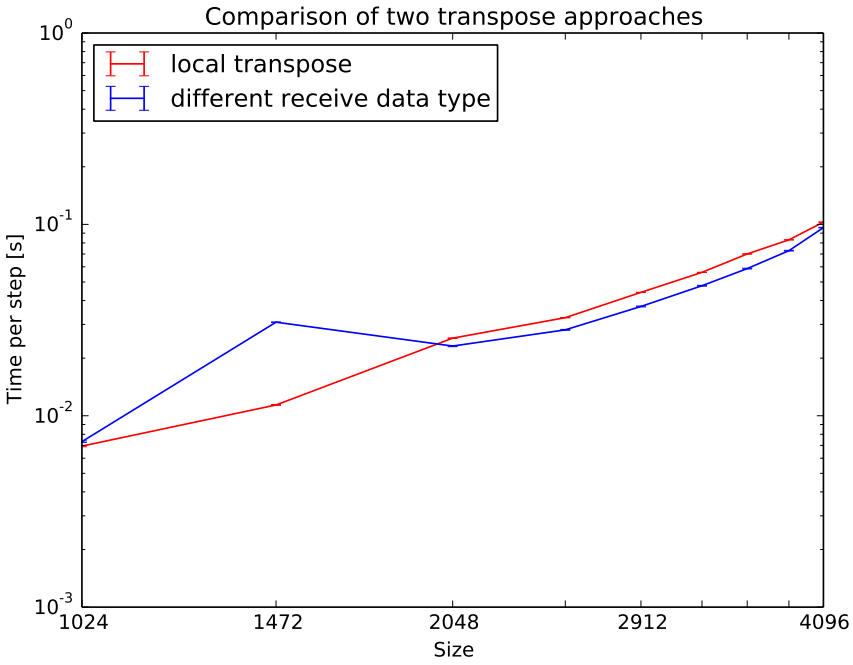

Figure 3: Comparison of the two approaches to the transpose after each half-step in the MPI versions with regards to their scaling with system size.

5.2.3 Different transpose approaches

In figure (3) we looked at the performance of our two different approaches to the transpose. We used the MPI version with 32 MPI processes and found that both approaches scale quite similarly.

5.2.4 Strong scaling

We also wanted to see how our implementation scaled without increasing the system size i.e. what the strong scaling plots look like.

First up the OpenMP version seen in figure (4) shows quite good speed up for the smaller system sizes but stagnates around 8 threads for all system sizes. This stagnation seems to be reasonable considering that in our test environment after 8 threads hyperthreading sets in.

We also think that we could have improved the performance of the OpenMP (and indirectly the hybrid version) by making it NUMA aware for instance by using a first-touch policy for our u and v vector.

The speed up of the pure MPI version (see figure (5)) on the other hand depends strongly on the size of the system. For smaller systems the speed

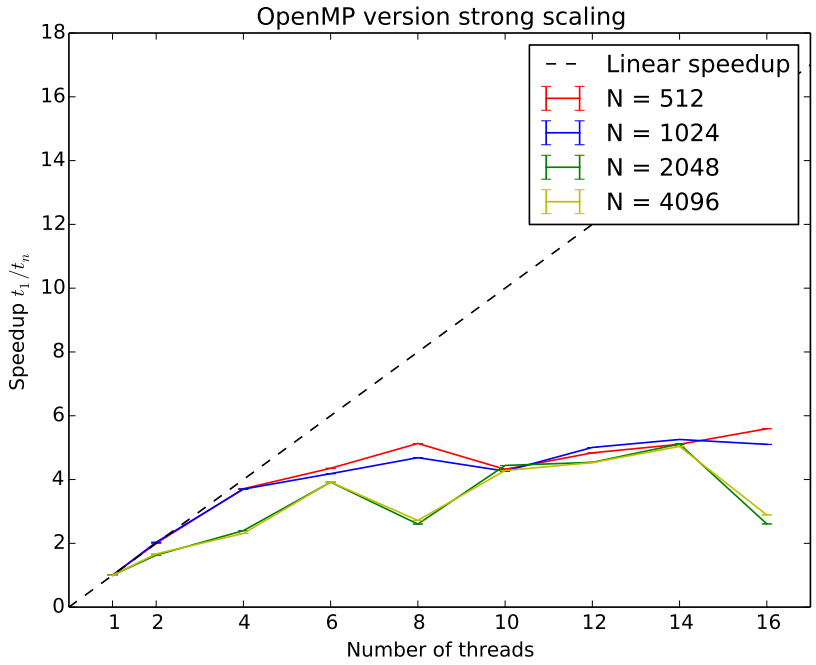

Figure 4: Strong scaling plot of the OpenMP implementation.

up stagnates around 16 processes while for the larger systems almost linear speed up is achieved up to 128 processes.

Lastly the strong scaling of the hybrid version (seen in figure (6)) shows similar structure as the pure MPI one. However for the larger systems it also shows super linear scaling up to 128 processes with 8 OpenMP threads each.

We also experimented around with the number of OpenMP threads in the hybrid version. (seen in figure (7)) We found that while it can make a difference in smaller systems and when not using many MPI processes, for the larger systems and when using more MPI processes the difference the number of OpenMP threads makes becomes less and less important to the overall scaling.

5.2.5 Energy consumption

During one of the strong scaling experiments for the pure MPI version we looked at the energy consumption reported by the Piz Daint system.
The results are summarized in table (**??**), as can be expected the energy

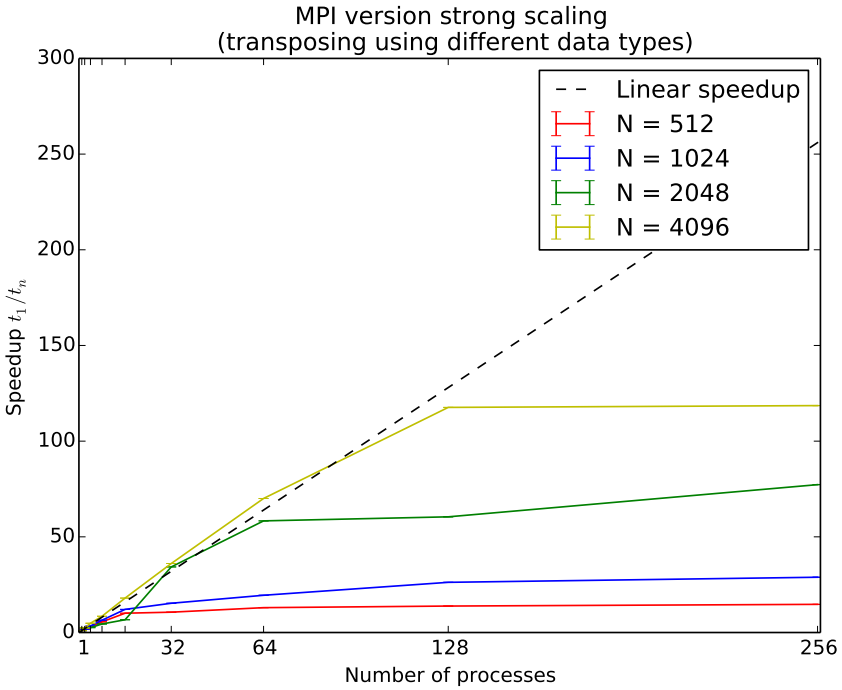

Figure 5: Strong scaling plot of the MPI implementation.

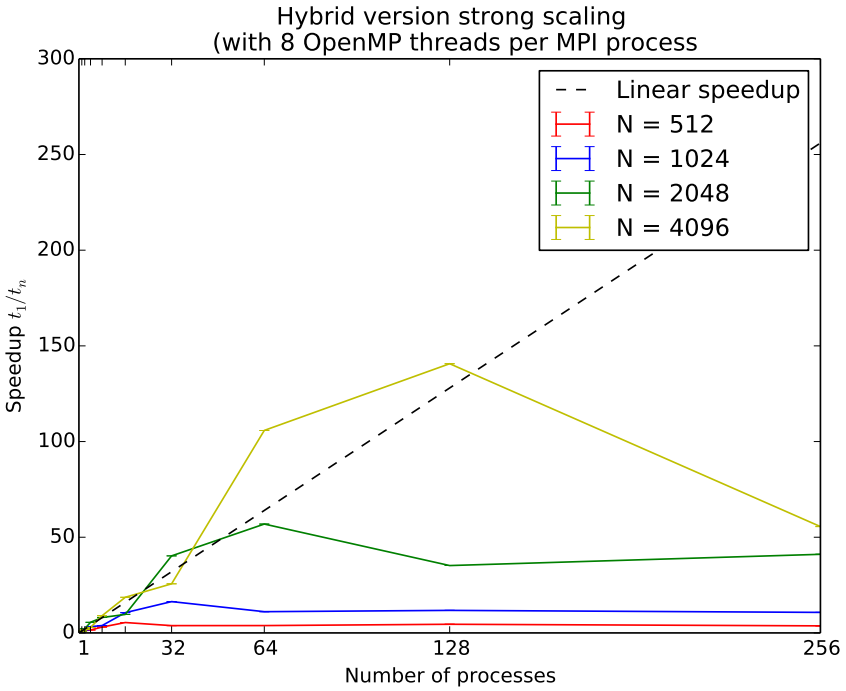

Figure 6: Strong scaling plot of the Hybrid MPI+OpenMP implementation.

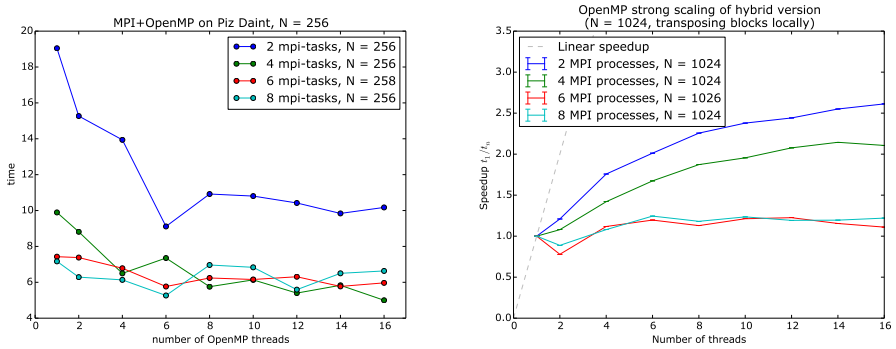

Figure 7: The Difference the number of OpenMP threads makes in the hybrid version for two example system sizes.

consumption varies due to how long the simulation takes and on how many nodes are used. Of course both of these factors depend on the number of processes used. In our case we can see a sharp decline in power consumption in the region we see good speed up in figure (5) and then a strong increase in power consumption, when the speed up stagnates.

MPI processes	1	2	4	8	16	32	64	128	256
Energy [in joule]	66982	30434	36037	25470	34324	13861	17199	32931	52283

Table 1: Energy consumption during strong scaling experiments.

6 Summary

References

[1] P Gray and SK Scott, "Autocatalytic reactions in the isothermal, continuous stirred tank reactor: isolas and other forms of multistability," *Chemical Engineering Science*, vol. 38, no. 1, pp. 29–43, 1983.

[2] I.J.D. Craig and A.D. Sneyd, "An alternating-direction implicit scheme for parabolic equations with mixed derivatives," *Computers & Mathematics with Applications*, vol. 16, no. 4, pp. 341–350, 1988.

[3] J E Pearson, "Complex patterns in a simple system.," *Science (New York, N.Y.)*, vol. 261, no. 5118, pp. 189–192, 1993.

[4] Wikipedia, "Tridiagonal matrix algorithm — Wikipedia, the free encyclopedia," 2014, [Online; Version from 9 April 2014].

[5] Lode Vandevenne, "Lodepng version 20140624," .