HPCSE PROJECT 2 - SOLVING AND OPTIMIZING A 2D DIFFUSION PROBLEM

Pierre Beckmann

ETH Zürich

ABSTRACT

In the contect of a High Performance Computing course of ETHZ we had to solve and optimize a computational engineering problem. This report summarizes the implementation and different steps of optimization that we went through with the diffusion project.

1. BACKGROUND

Diffusion describes a spreading processfrom a quantity from high density regions to low density regions. In this project we consider a two-dimensional domain medium described by the following equation:

$$\frac{\partial(\mathbf{r},t)}{\partial t} = D\Delta\rho(\mathbf{r},t)$$

with D the diffusion coeddicient and ρ the scalar quantity of interest at position r and time t. We used Dirichlet boundary conditions:

$$\rho(x,y,t) = 0 \qquad \forall x,y = \{0,1\}$$

and initial density distribution:

$$\rho(x,y,t) = \sin(\pi x)\cdot\sin(\pi y)$$

which yields the following analytical solution:

$$\rho(x,y,t) = \sin(\pi x)\cdot\sin(\pi y)\cdot e^{-2\Delta\pi^2 t}$$

To solve this continuous problem with numerical methods we do a discretization of space and time: $t_n = n\cdot\delta t$, $x_i = i\cdot\delta x$ and $y_j = j\cdot\delta y$. $\rho^n_{i,j}$ then represents the quantity of interest at position (x_i, y_j) and time t_n. We consider a uniform grid: $\delta h = \delta x = \delta y$.

We used two different methods to solve the diffusion euqtion.

1.1. Finite differences with Alternating Direction Implicit method (ADI)

ADI is a finite difference scheme that consists of the two following steps:

Step 1 Implicit Euler in x direction and explicit Euler in y direction:

$$\rho^{n+\frac{1}{2}}_{i,j} = \rho^n_{i,j} + \frac{D\delta t}{2}\left[\frac{\partial^2\rho^{n+\frac{1}{2}}_{i,j}}{\partial x^2} + \frac{\partial^2\rho^n_{i,j}}{\partial y^2}\right]$$

Step 2 Explicit Euler in x direction and implicit Euler in y direction:

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

Rewriting the first half step by applying the second order central difference on x we get:

$$\rho^{n+\frac{1}{2}}_{i,j}\left(\frac{D\delta t}{\delta x^2}+1\right) - \rho^{n+\frac{1}{2}}_{i-1,j}\left(\frac{D\delta t}{2\delta x^2}\right) - \rho^{n+\frac{1}{2}}_{i+1,j}\left(\frac{D\delta t}{2\delta x^2}\right) = \rho^n_{i,j} + \frac{D\delta t}{\delta x^2}\left[\frac{\partial^2\rho^n_{i,j}}{\partial y^2}\right]$$

Which in matrix form and second order central difference on y gives:

$$\mathbf{A}\rho^{n+\frac{1}{2}}_j = \rho_j + \frac{D\delta t}{2\delta x^2}(\rho^n_{j-1} - 2\rho^n_j + \rho^n_{j+1})$$

with:

$$\mathbf{A} = \begin{pmatrix} \frac{D\delta t}{\delta x^2}+1 & -\frac{D\delta t}{2\delta x^2} & 0 & \\ -\frac{D\delta t}{2\delta x^2} & \frac{D\delta t}{\delta x^2}+1 & -\frac{D\delta t}{2\delta x^2} & \\ \ddots & \ddots & \ddots & \ddots \\ & 0 & -\frac{D\delta t}{2\delta x^2} & \frac{D\delta t}{\delta x^2}+1 \end{pmatrix}$$

The same can be done with the second half step to get similar results.

For each half time step the tridiagonal system can then be solved using the Thomas algorithm: performing forward seep and update c'_i and d'_i for each value x_i of the solution vector and obtain the solution using back substitution $x_i = d'_i - c'_i x_{i+1}$. In this case the Thomas algorithm is simplified by the fact that a, b and c values are all the same in our A matrix.

This algorithm is unconditionally stable and second order in both space and time.

1.2. Random walks

With the random walks method we track the Brownian motion of a set of M equally weighted particles.

We use the following initial conditions:

$$m^0_{i,j} = \left\lfloor \rho(x_i, y_j, 0) \cdot \frac{M}{\iint_\Omega \rho(x, y, 0)\, dx\, dy} \right\rfloor$$

At each time step a particle can stay at its position with probability $1 - 4\lambda$ or move in the four relative directions with equal probability λ. In this case $\lambda = D\delta t/\delta x^2$ with the CFL condition $\lambda < 1/2$.

At each time step the approximation of the solution is given by:

$$\rho^n_{i,j} = \frac{m^n_{i,j}}{M} \iint_\Omega \rho(x, y, 0)\, dx\, dy$$

with $m^n_{i,j}$ the number of particles on the node.

This method is of order 0.5 with respect to the number of particles M.

1.3. Implementation and Optimization Tools

This project was coded in C++ using Advanced Vector Extensions (AVX) for vectorization, Open Multi-Processing (OpenMP) for shared-memory parallelization and Message Passing Interface (MPI) for distributed-memory parallelization.

2. BASELINE IMPLEMENTATION

We used the solution of a diffusion problem of the HPCSE class as a template. We created a `Diffusion2D` class which has the constants and parameters of our problem as attributes: `D`, `L`, `N`, `dr`, `dt`, `fac` for λ and the vector `rho` (of size `N`^{2}). $\rho^n_{i,j}$ is given by accessing the $(i * N + j)$th value of `rho`.

It also contains following methods:

- `initialize_density()` that initializes rho given the intial conditions.
- `advance()` that performs two ADI half steps to advance the state of the diffusion of one time step.
- `exact_rho()` that gives the exact solution $\rho(x_i, y_j, t)$ given x_i , y_j and t.
- A getter of rho and a function to write the resulting density into a file.

The outter lines of the ρ matrix are set to 0 and the computations are only done on the inner part.

A diffusion problem can then be started and advanced by the desired number of times in the main by using the `Diffusion2d` class. We offer the user two options while calling the executable: `-d` to solve the diffusion problem given certain parameters up to a chosen time and `-c` to get study the convergence of the error according to N for example.

2.1. ADI

ADI also contains the attributes `a`, `b`, `c` and vectors `c_p` and `d_p` used for the Thomas algorithm.

The two half steps were implemented using the Thomas algorithm. A `rho_tmp` is used to save the old version of `rho` while updating it. `std::swap` is used at the beginning of each half step because it is computationally more efficient than the equal operator.

The first half step operates over the lines of the matrix which means bad memory access for the neighbouring cells. A for loop iterates over the lines of the ρ matrix and, using the Thomas algorithm and the formula described in the background, we update c'_i and d'_i for each element of the line with a second for loop. Finally we use the c' and d' parameters with back substitution to get the half time solution line by line.

The second half step does the same iterations over columns which means better locality.

2.1.1. Convergence Analysis

The convergence analysis successfully yields a second order convergence in both space (Fig. 1) and time (Fig. 2) .

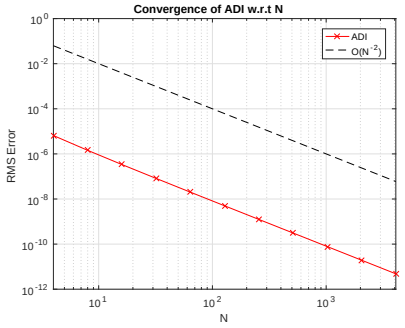

Fig. 1. Convergence of the ADI method with respect to N. The RMS error was computed for the same diffusion problem with eleven different N values. This is plotted in red. To visualize the order of convergence easily we plot second order in black.

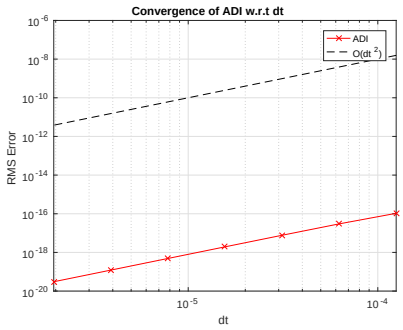

Fig. 2. **Convergence of the ADI method with respect to dt.** The RMS error was computed for the same diffusion problem with seven different dt values. This is plotted in red. To visualize the order of convergence easily we plot second order in black.

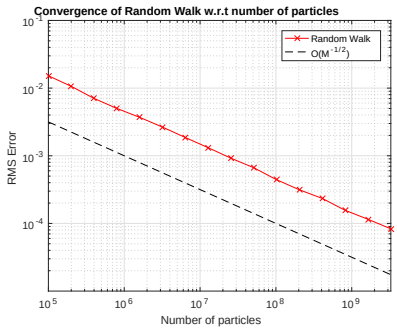

Fig. 4. **Convergence of the Random Walks method with respect to the number of particles M.** The RMS error was computed for the same diffusion problem with sixteen different M values. This is plotted in red. To visualize the order of convergence easily we plot 0.5 order in black.

2.1.2. Performance Analysis

To do the performance analysis of our code we used the Roofline model (Fig. 3). We observe that the operational intensities as well as the serial version of the code is compute bound when considering the serial peak performance roof but memory bound when considering multithreading and vectorizing. This justifies the use of vectorizing and multithreading.

The percentage of peak performance reached is 16.3 $\%$.

2.2. Random Walks

The random walks Diffusion2d class also has a `particles` attribute that is the same size than `rho` and contains a certain number of particles in each cell $i * N + j$. At each timextep we iterate over the cells of `particles` and use a binomial distribution with the local number of particles $m_{i,j}$ and λ to determine how many particles move in each direction. This is computationally more efficient than to iterate over all the particles in each cell.

We use a `do while` with the binomial distribution to get a valid number of moving particles: less than $m_{i,j}$. We also only move particles if $m_{i,j} > 0$.

At the end rho is updated according to the formula stated in the background.

2.2.1. Convergence Analysis

The convergence analysis successfully yields a 0.5 order convergence with respect to the number of particles M (Fig. 4).

3. OPTIMIZATION RESULTS

3.1. ADI

3.1.1. Scalar Optimizations

First we performed some scalar optimizations.

We use `initialize_thomas()` to initialize the Thomas algorithm to precompute the c_i's and a vector of values $denom$ that corresponds to one over the denominator part in the updating of d_i'. For each line in the first half step and each column in the second we now compute the c_i' and the denominator values only once instead of N times and do N multiplacations instead of N divisions (which costs less cycles).

We also unroll the outter loops eigth times (which after testing gave better results than four or twelve) to allow prefetching and pipeling and prepare for vectorization. Extra care was taken for the extra lines and extra columns in the case where N is not divideable by eigth.

The performance of the optimized version can be observed on the roofline model (Fig. 3). The percentage of peak performance reached is 41.4 $\%$ which is 2.5 better than the original serial version.

3.1.2. Vectorization

Next we used AVX and intel intrinsics instructions to do vectorization. We used `__m256d` vectors of four to do operations simultaneously.

The way `rho` is implemented makes neighbours in the line right next to the cell but neighbours in the column are at a N values distance. This means better data locality when iterating over columns because we use data in the same line for the computations. Therefore the AVX implementation for the second half step was pretty straightforward.

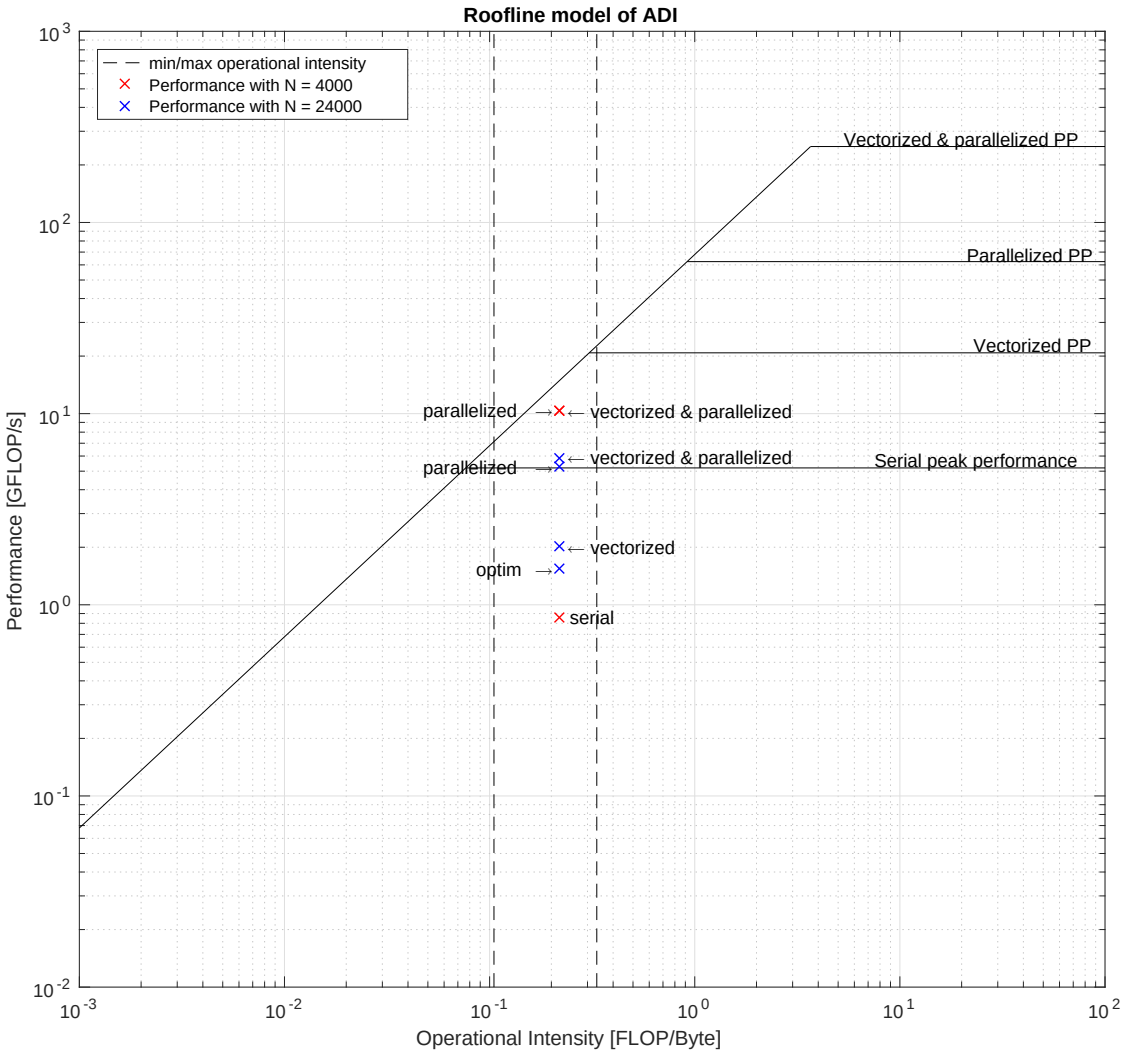

Fig. 3. **Roofline Model for ADI.** Red crosses correspond to a $N = 4000$ (with 1000 timesteps) size and blue crosses correspond to a $N = 24000$ size (with 40 timesteps). Those two different sizes yielded interesting results justifying their presence in this plot. For calculations we used the specifications of the XC50 Compute Nodes of the Piz Daint super computer: peak bandwidth $68GB/s$, core frequency $2.6GHz$, 2 FLOP per cycle, 12 cores per node and 4 floats for the SIMD width. Per timestep we counted 32 operation for the serial version, 22 for the optimized and 20 for the vectorized (fmadd removes operations). We also counted 38 memory accesses (32 reands and 6 writes) with no cache and 12 (6 reads and 6 writes) with infinite cache. Calculated minimal operational intensity is 0.105 $FLOP/Byte$ and maximal is 0.333 $FLOP/Byte$.

Vectorizing the first half step using gather and scatter operations was considered. However after implementing a version with gather instructions the code got significantly slower and we couldn't imagine it yielding a significant speedup even with the scatter instructions and we decided to abandon that option.

We wrote a version with unaligned instructions and another where we used a padded matrix and `_mm_malloc` to allocate data and aligned instructions. We observed no difference in speed between those two codes.

The performance of the vectorized version can be observed on the roofline model (Fig. 3). The percentage of peak performance reached is 13.6 % with N=24'000 and 40 time steps.

We observe that the unrolled version is slightly faster than AVX for low values of N (in blue), but AVX becomes significantly faster when increasing N (in red). We expected AVX to be faster in every situation. With small N the compiler is maybe better with prefetching and pipelining than with an implemented vectorization that puts constraints on the system.

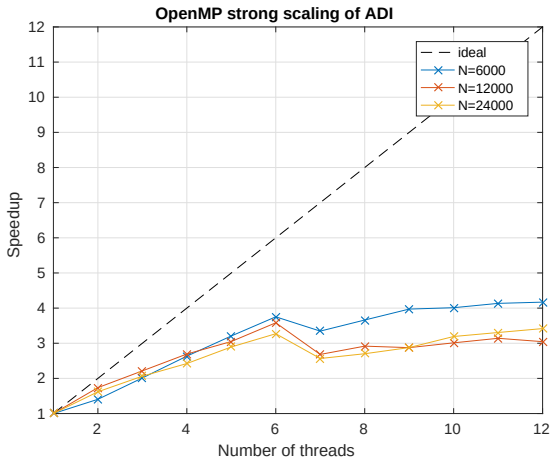

Fig. 5. OpenMP strong scaling of ADI. We plotted the speedup (timings ove initial timing) in function of the number of threads. On piz daint a node has twelve CPUs so we used one to twelve threads. We compare the ideal behavior to N=6'000, N=12'000 and N=24'000.

3.1.3. Shared-memory Parallelization

Shared-memory parallelization was done using OpenMP.

We decided to change the implementation and give the diffusion2d class a `tmax` parameter to change the advance function to a run function that performs all the timesteps. This allows the code to reuse threads instead of creating (and destroying) new ones at every timestep.

Parallelizing starts with spawning n threads at the beginning of the run function with an omp parallel block. The outter for loops of both timesteps are then run parallely by each thread with the omp for instruction. We had to change the d' vector to a variable in the run function rather than an attribute to make it firstprivate as each thread needs its own. Swaping between `rho` and `rho_tmp`, iterations for extra lines and extra columns and incrementation of time are done by only one thread using the omp single instruction. The nowait instruction added to the omp for allows threads that have finished their job to do the single instruction while the other threads finish their jobs.

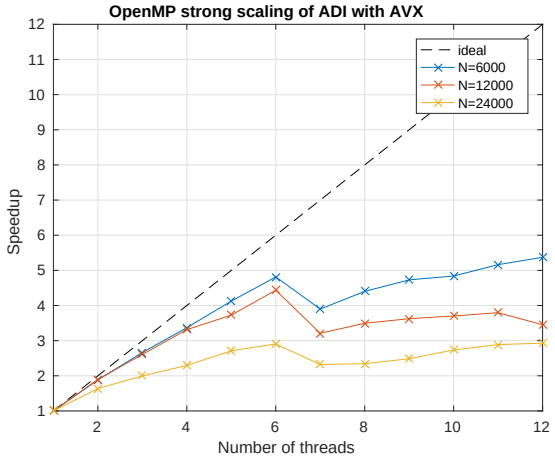

Fig. 6. OpenMP strong scaling of ADI with AVX. We plotted the speedup (timings ove initial timing) in function of the number of threads (one to twelve). We compare the ideal behavior to N=6'000, N=12'000 and N=24'000.

Scaling ADI scales with the number of threads when plotting the speedup and efficiency versus the number of threads up to twelve threads.

In Fig. 5 and Fig. 6 we see that OpenMp scales with and without AVX. Overall it seems to scale better with AVX and with a smaller grid size N. However we observe a strange behavior going from six threads to seven with a sudden speedup drop followed by a flatter scaling all the way up to twelve threads. This dip is very weird and also present in the random walks implementation which makes us think that it isomething related to piz daint and not (only) our code.

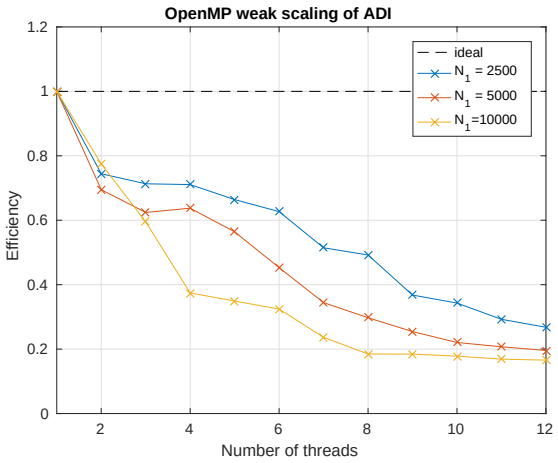

Fig. 7. **OpenMP weakscaling of ADI.** We plotted the efficiency in function of the number of threads (one to twelve). We compare the ideal behavior to initial sizes N_1=2'500, N_1=5'000 and N_1=10'000. Using $N_n = \sqrt{n}\, N_1$.

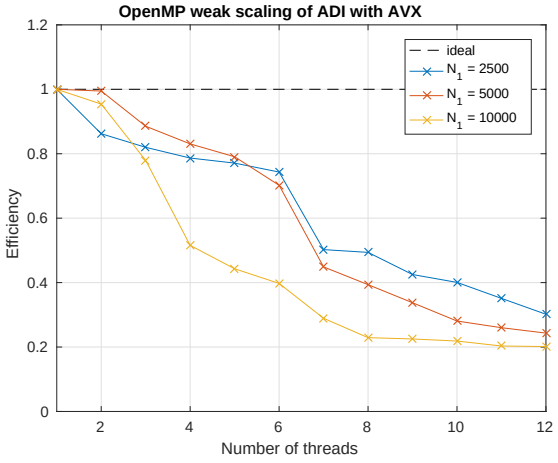

Fig. 8. **OpenMP weak scaling of ADI with AVX.** We plotted the efficiency in function of the number of threads (one to twelve). We compare the ideal behavior to initial sizes N_1=2'500, N_1=5'000 and N_1=10'000. Using $N_n = \sqrt{n}\, N_1$.

Weak scaling yields similar results. The efficiency for two threads is really good with two threads expecially with N_1=5000

The performance of the multithreaded version can be observed on the roofline model (Fig. 3). The percentage of peak performance reached is 69.6 % with N=4'000 and 1'000 time steps and 35.3 % with N=24'000 and 40 time steps.

3.1.4. Distributed-memory Parallelization

An unsuccessful attempt at an MPI implementation was done. The idea was to cut the ρ matrix into stripes and use an all-to-all instruction with blocks between the two half steps. However after a lot of testing we didn't got it to work.

3.2. Random Walks

3.2.1. Scalar Optimizations

To speedup the serial implementation we used a Intel Math Kernel Library (MKL) method to generate four binomial distributed random numbers simultaneously. It seems to make the code approximately 9.8 times faster (Fig. 9).

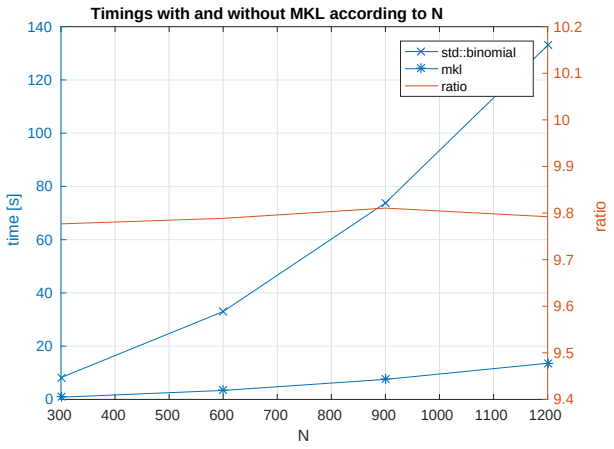

Fig. 9. **Comparing runtimes with and without MKL according to N.** The timings are in blue w.r.t. the right axis. The ratio of the timings is in yellow w.r.t. the left axis.

3.2.2. Vectorization

We couldn't think of a way to use vectorization for the random walks method.

3.2.3. Shared-memory Parallelization

Similarly to ADI we changed the advance function to a run function. This also allows to update `rho` according to `particles` only once at the very end. At the beginning of the function we spawn threads using a omp parallel block. We then save the moving particles in a `moves` vector that is four times the size of `particles`. This way we can parallelize the outter for loop which wouldn't have been possible using `particles` because of race conditions. As we have two nested for loops we tried collapse(2) but got no speedup. Then we update `particles` using `moves` in a single block followed by the also single time incrementation.

Scaling Random walks scales with the number of threads when plotting the speedup and efficiency versus the number of threads up to twelve threads.

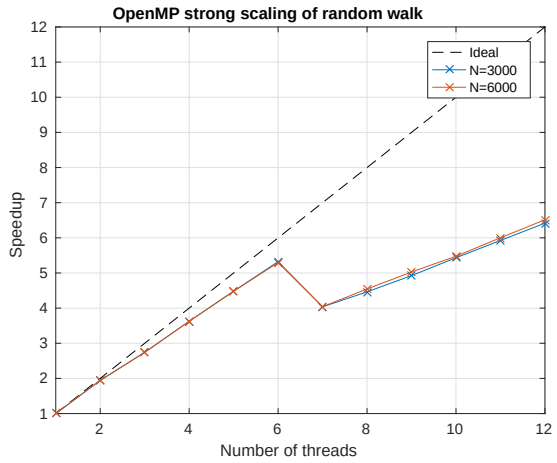

Fig. 10. **OpenMP strong scaling of Random Walks.** We plotted the speedup (timings ove initial timing) in function of the number of threads (one to twelve). We compare the ideal behavior to N=3'000, N=6'000.

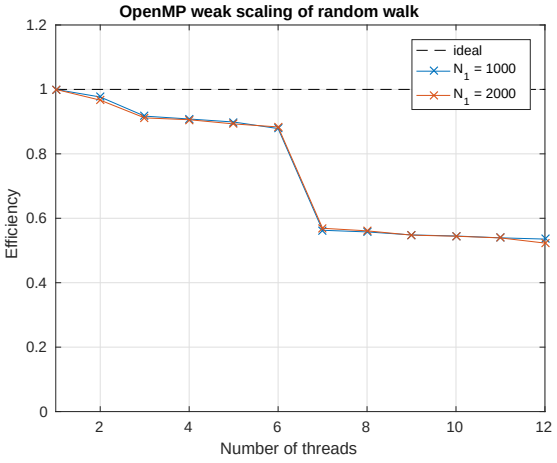

Fig. 11. **OpenMP strong scaling of Random Walks.** We plotted the efficiency in function of the number of threads (one to twelve). We compare the ideal behavior to initial sizes N_1=1'000 and N_1=2'000. Using $N_n = \sqrt{n}\, N_1$.

Strong scaling and weak scaling both indicate good scaling for the random walk method. However we observe the same strange dip again. We also observe values that fluctuate lass than for ADI and the same scaling for different grid sizes N.

3.2.4. Distributed-memory Parallelization

We used MPI to do distributed-memory parallelization.

Several processes are spawned in the main function. A diffusion2d class now contains a stripe (severall lines) of the complete problem with two ghost lines added respectively at the top and the bottom of the stripe.

Particles that diffuse onto ghost cells were meant to move respectively on the previous or next stripe. Therefore a swaping of ghost lines is done after each time step using `MPI_Isendv` and `MPI_Irecv` and a waitall operation. The swaping consists of exchanging our upper ghost line with the previous's lower ghost cell and exchanging your lower ghost line with the next's upper ghost line. After swaping we have to add the new ghost line onto the boundary line.

Scaling Random walks scales with the number of threads when plotting the speedup and efficiency versus the number of threads up to 36 threads.

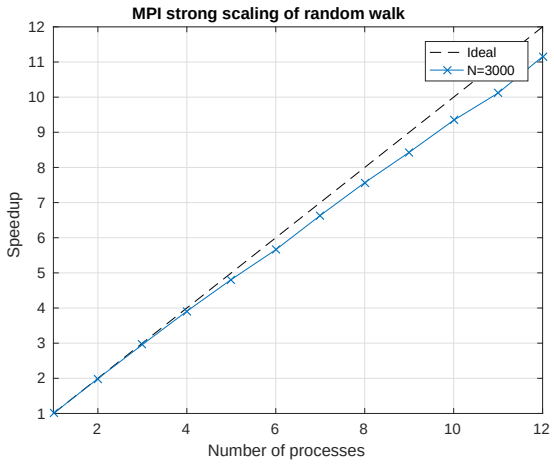

Fig. 12. **MPI strong scaling of Random Walks.** We plotted the speedup (timings ove initial timing) in function of the number of processes (one to twelve). We compare the ideal behavior to N=3'000.

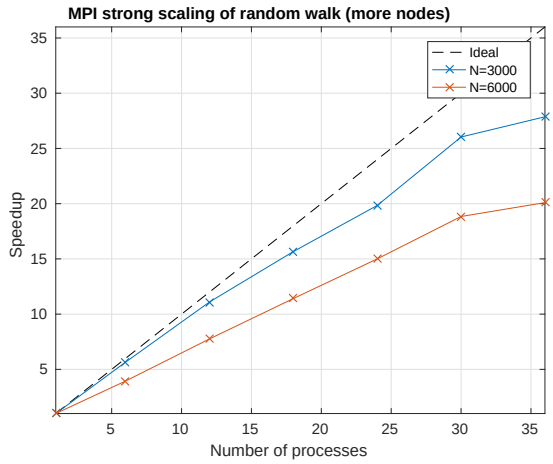

Fig. 13. **MPI strong scaling of Random Walks with more processes.** We plotted the speedup (timings ove initial timing) in function of the number of processes (one to 36 going six by six). We compare the ideal behavior to N=3'000 and N=6'000.

In Fig. 12 we observe a very good scaling of random walks according to the number of processes. Going up to more processes and comparing to a larger grid size N in Fig. 13 we observe a better scaling for smaller N and an unstable behavior starting at thirty processes.

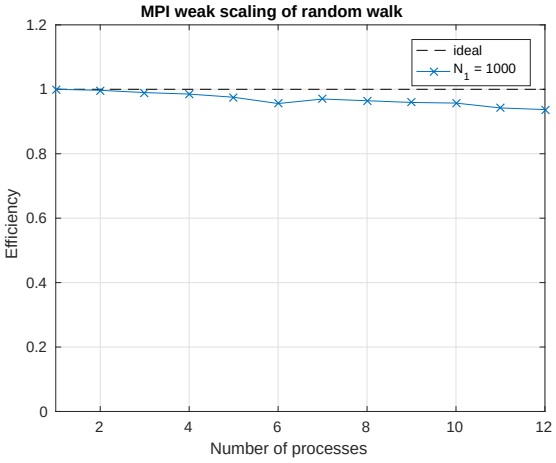

Fig. 14. **MPI weak scaling of Random Walks.** We plotted the efficiency in function of the number of processes (one to twelve). We compare the ideal behavior to N_1=1'000. Using $N_n = \sqrt{n}\, N_1$.

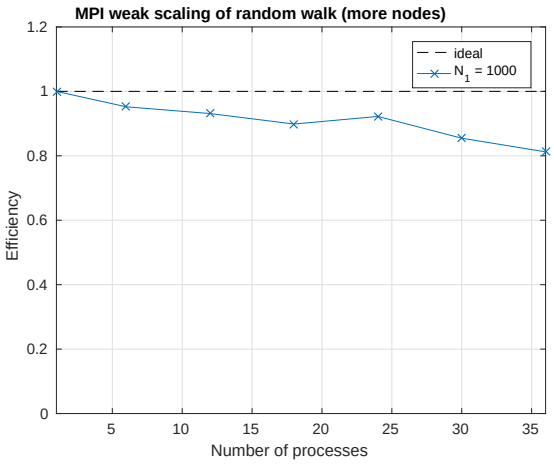

Fig. 15. **MPI weak scaling of Random Walks with more processes.** We plotted the efficiency in function of the number of processes (one to 36 going six by six). We compare the ideal behavior to N_1=1'000. Using $N_n = \sqrt{n}\, N_1$.

The weak scaling shows similar results.

4. CONCLUSIONS

For ADI the best optimization was parallelizing the code with OpenMP. Vectorization was less effective than expected probably because the compiler does better optimizations with the unrolled version and our implementation added some constraints.

For random Walks we got very nice results with MPI. The few communications make it very adapted for it.