HPCSE PROJECT #2: DIFFUSION

Mathis Lamarre

CSE
ETH Zürich
Zürich, Switzerland

ABSTRACT

This report presents our project for the HPCSE I and II class. We analyzed a diffusion problem and solved it with two methods : finite differences with Alternating Direction Implicit (ADI) and Random Walk (RW).

1. BACKGROUND

Diffusion is a very relevant process in many scientific and engineering fields. It can be described by the following equation:

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

with D the diffusion coefficient, ρ the density at position $\mathbf{r}$ (x and y in our two-dimensional case) at time t. We use Dirichlet boundary conditions:

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

and initial density distribution:

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

giving the analytical solution:

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

As for any numerical scheme, we need to discretize. We denote $t_n = n\cdot\delta t$ the time indices and $x_i = i\cdot\delta h$, $y_j = j\cdot\delta r$ the mesh points. The density at point (x_i, y_j) and time t_n is denoted as $\rho^n_{i,j}$.

1.1. Finite differences with Alternating Direction Implicit (ADI)

In this finite difference scheme, we alternate two half-steps. First, implicit Euler in x direction and explicit Euler in y direction:

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

Then, explicit Euler in x direction and implicit Euler in y direction:

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

Each half-step can be solved using the Thomas algorithm. We apply a central difference for the second order derivative on x and factorize the implicit terms:

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

We can then write it in the matrix form (with central difference for the second order derivative in y) :

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

with the tridiagonal matrix:

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

Thus, a half-step consists of solving m tridiagonal systems of equation with m the size of the grid. In the first one, we solve for each line and and the second for each column. This scheme is unconditionally stable in space and time.

1.2. Random Walks

We simulate the Brownian motion of M equally weighted particles without inertia. This is the initial condition:

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

with $m^n_{i,j}$ the number of particles on cell (x_i, y_j) at time t^n. At each time step, each particle remains at the same position with probabilty $1 - 4\lambda$ and moves in either of the four possible directions with probability λ ($\lambda = \frac{D\delta t}{\delta x^2}$). Then, the density is updated as follows:

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

1.3. Tools and optimization

All the code was written in C++. To optimize, OpenMP, MPI as well as AVX and MKL were used.

2. BASELINE IMPLEMENTATION

2.1. ADI

2.1.1. Code-base structure

To represent our problem, we take advantage of object-oriented programming and use a `class` called `Diffusion2D`. Its attributes are: the constants (`D_`,`L_`), the size of the problem (`N_`,`Ntot_` and corresponding `dr_`), the time step (`dt_`) and the factor `fac_` which represents λ. It also has vectors `rho_` and `rho_tmp` to contain the density value at each mesh point. Finally, it has the values `a_`, `b_`, `c_` and vectors `c_p` and `d_p` used for the Thomas algorithm. All these values are initialized in the constructor.

The main part is the function `advance()`. It computes the density `rho_` at the following time step. To do so, it uses the Thomas algorithm on all the lines (first half-step) and then on all the columns (second half-step). Each half-step is preceded by a `swap` of `rho_` and `rho_tmp` to put the old values in the temporary vector (more efficient than `rho_tmp=rho_` because it does not have to copy).

In the first half step, the outer loop is on `i` between `1` and `N_-2`. We don't iterate over the first and last line because they are always equal to zero (Dirichlet boundary conditions). We first calculate the first d' coefficient of the Thomas algorithm.

Then, in an inner loop on the `j`, we calculate the c' and d' coefficients. Here, the right-hand side is not simply the vector `rho_tmp`, but rather a sum and subtraction of the values and the left and right side (see formula derived in the background). Note that the coefficients a, b and c do not need a vector (unlike d, c' and d') because, in our case, they are all the same.

Then, we calculate the last element d' and set it right away to the last value of `rho_` in the line `i` (second-to-last actually, because the last is zero). Finally, in a second inner loop, we iterate backwards on all the `j` and set the values of `rho_` on the line `i` using the vectors computed above.

Then, we do the second half-step. It proceeds in the same but the outer loop is on the columns `j` and the inner loops are on the lines `i`.

This function is called in a `while` loop in the main, by incrementing the `time` by `dt_` at each iteration, until `tmax` is reached.

2.1.2. Verification

In order to verify our algorithm, we ran a convergence analysis both in space and time. The order of convergence should be 2, in both cases. As we can see in fig.1 and fig.2, it is indeed the case.

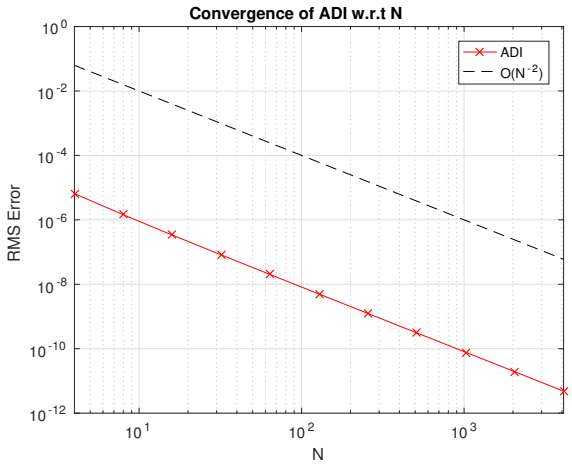

Fig. 1. Convergence of the ADI implementation with respect to the grid size N (space). The red line is the RMS error. The black line is the ideal order of convergence of 2.

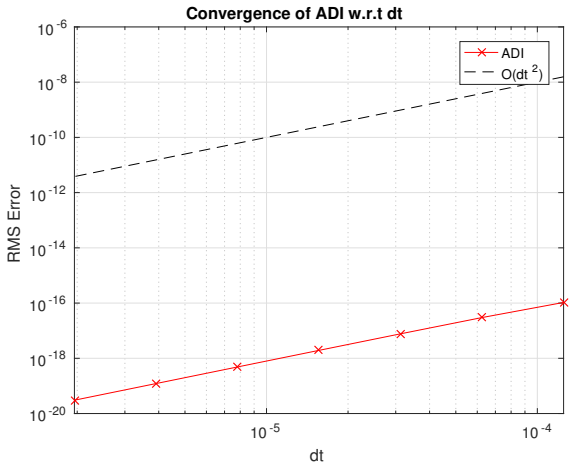

Fig. 2. Convergence of the ADI implementation with respect to the time step dt (time). The red line is the RMS error. The black line is the ideal order of convergence of 2.

2.1.3. *Performance analysis*

We used the roofline model (see fig.3). The Peak Performances and Peak Bandwidths corresponds to the Compute Nodes of Piz Daint. The number of floating point operations as well as memory operations were calculated manually. As we can see, when taking into account only the serial and scalar peak performance, we are compute bound. This justifies the use of vectorization and multithreading. However, when we consider vectorization, parallelization and both, we are memory bound.

In the serial implementation, we reach 16.4% of the peak performance.

2.2. RW

2.2.1. *Code-base structure*

Here again, we implemented a `Diffusion2D` class. Its attributes are the same constants as for ADI, with addition of `M_` the number of particles and without the coefficients of the Thomas algorithm. It also has a `rho_` vector as well as `particles_` and `particles_tmp_` vectors which contain the number of particles at each mesh point ($m^n_{i,j}$), so a `size_type`.

The main most important part is the function `advance()`. It moves the particles on all the cells and updates the density. To simulate the Brownian motion, instead of drawing a random number for each particle, we use a binomial distribution to describe how many particles on each cell move in a certain direction. This way, instead of computing $m^n_{i,j}$ random numbers on each cell, we can compute only 4.

To get these random numbers, we use a `mt19937` generator whose seed is different at each call with a `random_device`. We iterate over the entire grid with 2 for loops (`i` and `j` from `1` to `N_-2`) with the exception of the edges because of the boundary conditions.

Then, we call the binomial distribution 4 times with `particles_[i*N_+j]` as the number of trials and `fac_` as the probability of success. We increase the number of particles on the neighboring cells with the random numbers and decrease the number of the present (while making sure it remains positive).

These changes are made on the vector `particles_tmp_` as we want the number of particles on the other cells to remain constant while we iterate (so that a particle can only move once at each time step). Finally, we update the density `rho_` with the new number of particles.

2.2.2. *Verification*

To make sure our implementation is correct, we ran a convergence analysis. With such a Monte-Carlo method, the estimated order of convergence is $\frac{1}{2}$. As seen on fig.4, it is indeed the case.

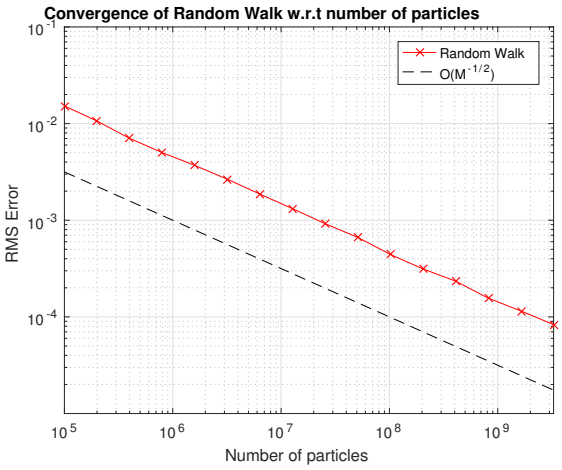

Fig. 4. Convergence of the RW implementation with respect to the number of particles M. The red line is the RMS error. The black line is the ideal order of convergence of $\frac{1}{2}$.

3. OPTIMIZATION RESULTS

3.1. ADI

3.1.1. *Scalar optimizations*

The first strategy we applied is scalar optimization. As the vector `c_p` is always the same (it does not depend on `rho_` or the time), we precompute it. Also, as the denominator in the formula of the `c_p`, `d_p` and `rho_` is constant, we precompute it and store as its inverse. It is in the vector `denom`. This way, we will use it in multiplication as opposed to division which are much more costly.

We also unrolled the outer loop so that the compiler can perform pipelining and prefetch operations. This could also enable the compiler to vectorize, particularly in the second half-step. As each iteration is on a line, the memory accesses are much closer: `rho_tmp[i*N_+j]`, `rho_tmp[i*N_+j+1]` in contrast to `rho_tmp[i*N_+j]`, `rho_tmp[(i+1)*N_+j]`. We added a section that performs the algorithm on the "extra-lines/columns", since the number of `N_` might not be a multiple of the unrolling factor.

This improved the performance as we see a higher plot for *optim* than *serial* on fig.3. It reaches 41.4% of the peak performance, which is 2.5 times better than the serial version (fig.3).

3.1.2. *Vectorization*

We vectorized the code using AVX. First, we did it only in the second half-step, as the memory accesses are consecutive in the vector. We use `__m256d` registers and `_mm256_loadu_pd` and `_mm256_storeu_pd` to get and set data in the vectors. We were also able to reduce the number of operations using

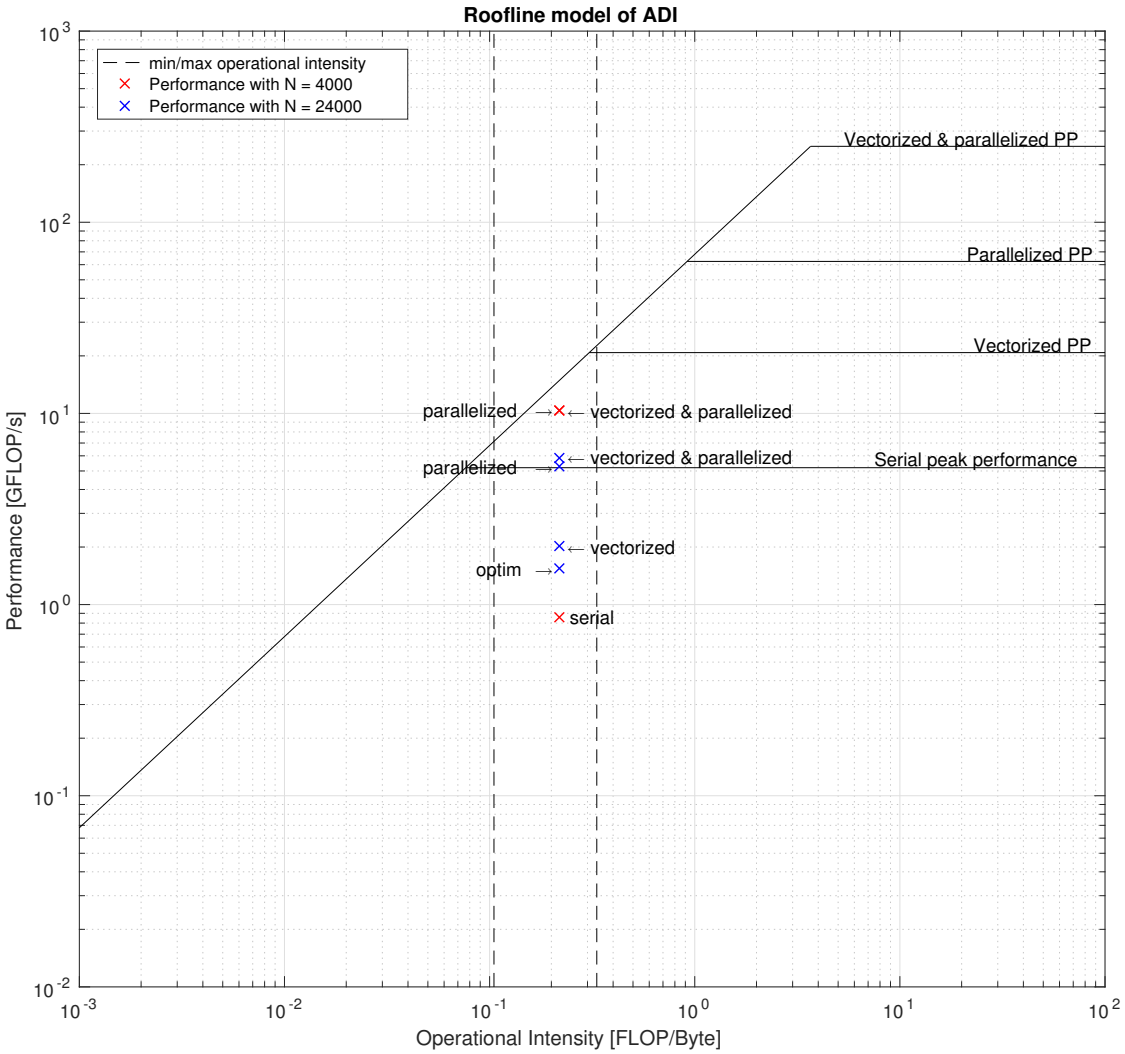

Fig. 3. Performance analysis of ADI on the roofline model. To calculate the various peak performances, we used the specifications of the XC50 Compute Nodes of the Piz Daint super computer of the CSCS. The core frequency is 2.6 GHz and the number of FLOP computed in each cycle is 2. The SIMD width is 4 floats and the number of cores is 12. The peak bandwidth is 68 GB/s. We counted 32 floating point operations for the serial implementation, 22 for the optimized version and 20.5 for the vectorized one. We counted 38 memory accesses (32 reads and 6 writes) with no cache and 12 memory accesses (6 reads and 6 writes) with infinite cache. The minimal and maximal operational intensity are thus 0.105 $FLOP/Byte$ and 0.333 $FLOP/Byte$. The performances were computed with 1000 times steps for $N = 4000$ and 40 time steps for $N = 24000$.

_mm256_fmadd_pd which performs a multiplication and a addition at the same time.

With small grid sizes, the results were similar to the scalar optimization, yielding slightly better runtimes but equivalent performance if we consider the number of operations. However, with much bigger grid sizes, the performance did go up.

Then, we tried to vectorize the first half-step, where the accesses in the rho_tmp vector are N_ appart. To do so, we used _mm256_i32gather_pd but we were not able to find a scatter function that worked. The results were very slow, which indicates that our way of vectorizing was not optimal.

We tried to align the data to make the vectorizing more efficient. This would allow the use of _mm256_load and _mm256_store instead of their unaligned version, which are slower. To do so, we padded the matrix so that the the length of a line/column is a multiple the number of doubles a register can hold. We used _mm_malloc to align the memory blocks. The results were mixed as the runtime was unaffected.

The performance reaches only 13.6% (fig.3).

3.1.3. Parallelization

Shared-memory. We parallelized the code with OpenMP. We changed the advance() function (which did only 1 time step and was called in a while loop) to a run() function which is only called once and compares the time to tmax_ itself. This way, the threads are spawned and joined only once instead of doing it at each time step. The entire while is in a omp parallel section.

We also had to do changes concerning race conditions: the vector d_p_ could not be a class member as it has to be firstprivate (because it was already initiated to the correct size). Each threads will work on different lines/columns, so they must each have a copy of it.

The swap is done in a single section, so that it is done only once (more times is unnecessary and an even time would make no swap at all).

The outer for loop is parallelized with a omp for instruction. This way, each thread will do a portion of the loop. rho_tmp is shared since it is only read. rho_ is also shared because no threads write on the same line. We added a nowait because this section is followed by a omp single section which can be done independently.

The "extra lines" are done in a omp single section because there are only a few (up to 3 or 7). This section as an implicit barrier because it is followed by a the swap, so all the works needs to be done (threads synchronized).

In the second half-step however, the omp single of the "extra columns" has a nowait clause because it is followed a by the time incrementation which is independent (also in a omp single section so that it's done twice). The implicit barrier at the end of this section ensures that the threads will be synchronized before the start of the next iteration.

As can be seen on fig.3, the performance is way higher with the parallelization. With $N = 4000$, we achieve 69.6% of the peak performance, but only 35.2% with the bigger problem size.

We also performed weak and strong scaling both with and without AVX vectorization. The strong scaling is pretty good from 1 to 6 threads (fig.5). However there is a dip in speed at 7 threads. When using AVX, the scaling is slightly better with small sizes but worse with big N (fig.6).

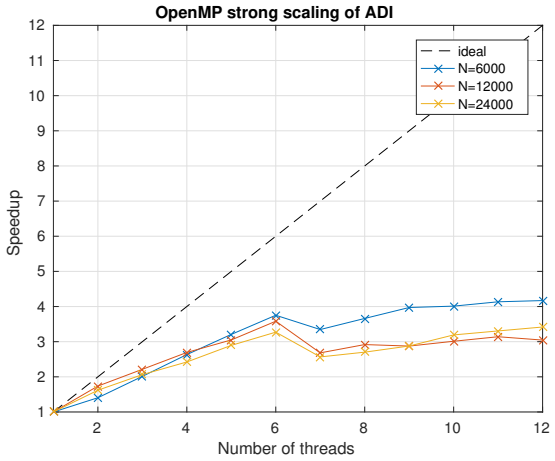

Fig. 5. Strong scaling of ADI with OpenMP multithreading: constant N and increasing threads.

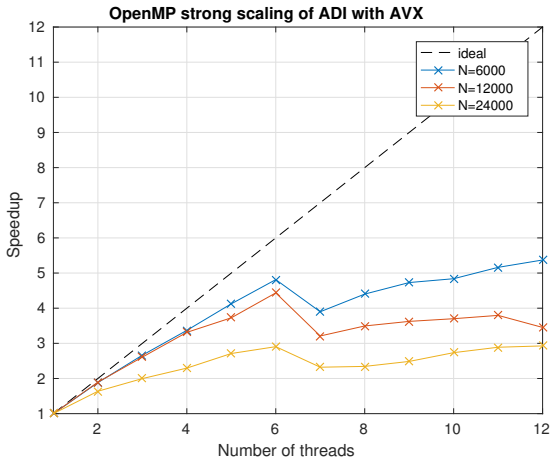

Fig. 6. Strong scaling of ADI with OpenMP multithreading and AVX vectorization: constant N and increasing threads.

Concerning weak scaling, the results are similar (fig.7)

with the exception of 2 threads with AVX vectorizing (fig.8) which give an efficient close to 1.

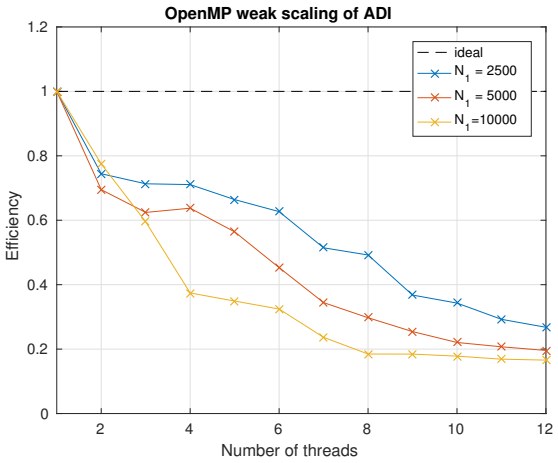

Fig. 7. Weak scaling of ADI with OpenMP multithreading: N^2 proportional to number of threads.

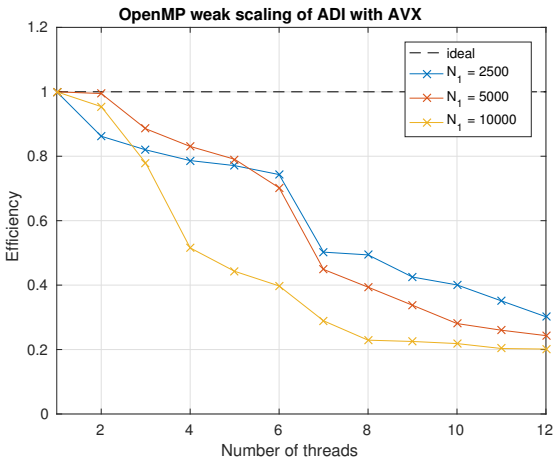

Fig. 8. Weak scaling of ADI with OpenMP multithreading and AVX vectorization: N^2 proportional to number of threads.

MPI. We also tried to parallelize with the Message Passing Interface (MPI). However, we could not find a domain decomposition that worked. In fact, the first half-step we want each process to have a line-wise portion of the grid, but a column-wise portion in the second half-step. We tried to use the function `MPI_Alltoall` but without success.

3.1.4. Parallelization and vectorization

We parallelized the vectorized code. As previously stated, its performance is similar with small sizes but better with big sizes (fig.3).

3.2. RW

3.2.1. Scalar optimization

To reduce the number of times the random number generator is called, we used `viRngBinomial` from the Intel Math Kernel Library (MKL). It can generate several number from a binomial law at once, so we can make 1 call instead of 4.

As we can see on fig.9, the MKL version is way faster (almost 10 times), independently of the grid size N.

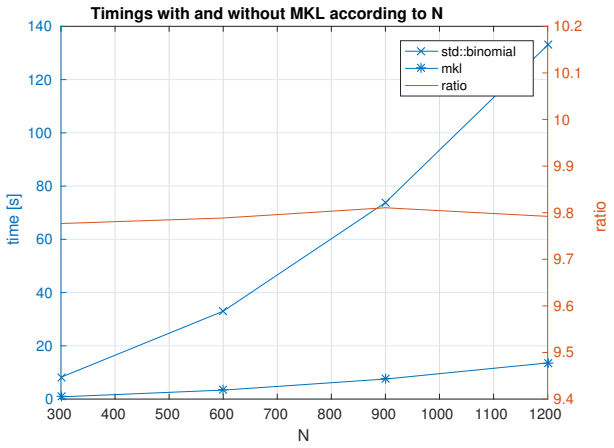

Fig. 9. Comparison of runtimes of RW with `std::binomial` and `viRngBinomial` from the MKL. Ratio is given with different scale.

3.2.2. Parallelization

Distributed-memory. In order to parallelize with MPI, we need to add 2 attributes to our class `Diffusion2D`: `rank_` and `procs_`, both `size_type`. Then, we compute `local_N_` by dividing `N_` by the number of processes. We make sure that the last process has more lines in case the grid size is not a multiple of the number of processes. Also, we add to ghost cells to each process: they will be used to compute and communicate the particles that move on the other processes' domain. Then, in the `run()` function, we do not iterate over the entire domain, but only on the `local_N_` lines that belong to each process. Furthermore, we skip the first (`i = 0`) and last (`i = local_N+1`) lines because they correspond to the ghost cells. For the first and the last process, we can skip lines before and after that, because they are on the edges.

We compute the random numbers as before and increment `particles_tmp_` accordingly.

Next, we swap the ghost lines with other processes: they contain the number of particles that moved on each process'

first and last line from the domains above and below. We do so using `MPI_Irecv` and `MPI_Isend`, sending the first and last line and requesting data on these same first and last line. We finish these communication with a `MPI_Waitall` call, to make sure that all ghost lines arrived before adding.

We add the received ghost lines of the boundary lines. We then set them back to 0 so that, at each time step, they contain just the number of particles that newly moved on them.

We tested the scaling of this implementation. As we can see on fig.10, it is very good, much better than with OpenMP. The speedup is almost ideal. We also did it with more nodes (fig.11, which is impossible with OpenMP (we could only go up to 24, and that's with hyperthreading).

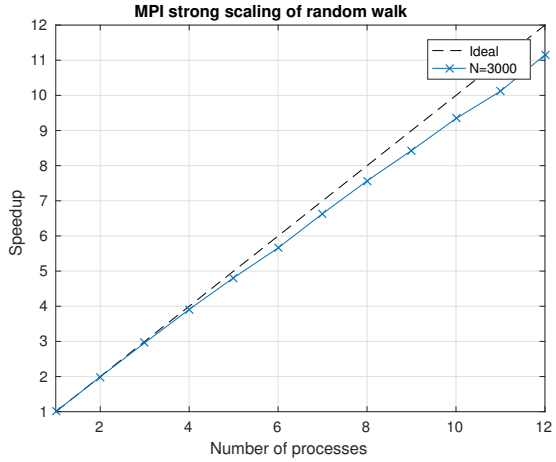

Fig. 10. Strong scaling of RW with MPI from 1 to 12 nodes. Constant N and increasing number of processes.

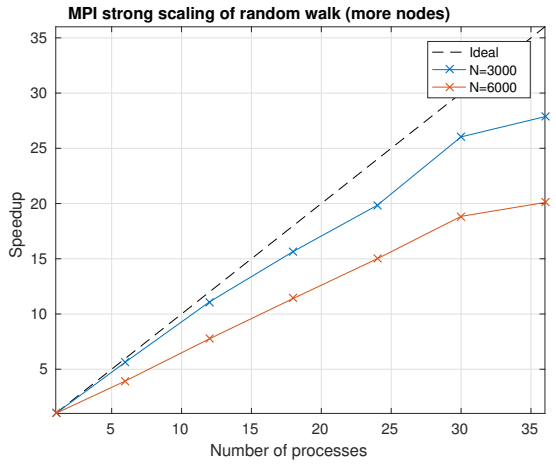

Fig. 11. Strong scaling of RW with MPI from 1 to 36 nodes. Constant N and increasing number of processes.

The weak scaling also yields satisfying results (fig.12 and fig.13).

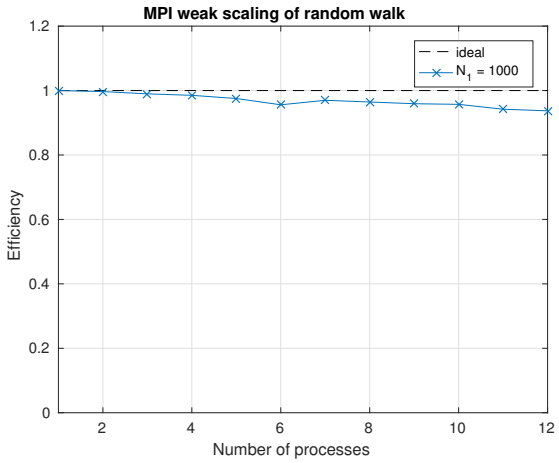

Fig. 12. Weak scaling of RW with MPI from 1 to 12 nodes. N^2 proportional to number of processes.

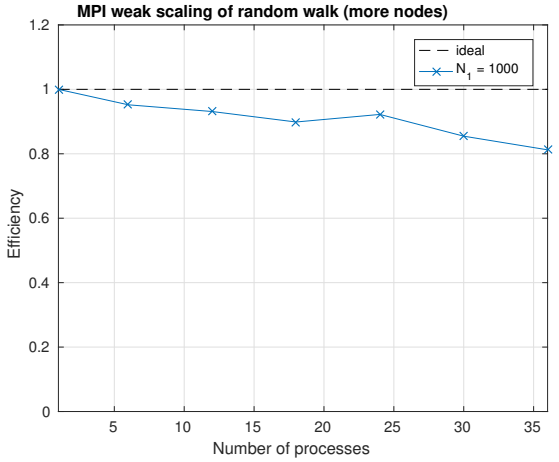

Fig. 13. Weak scaling of RW with MPI from 1 to 36 nodes. N^2 proportional to number of processes.

Shared-memory. To multithread the code with OpenMP, we need to be careful about race conditions on `particles_tmp_`. All the thread need to read and write on it at the same time, and sometimes in the same location! To remedy this, we slightly change the algorithm.

We create a vector `moves_`, 4 times the size of `particles_`, that is going to hold the 4 random number for the direction on each cell. Then, in a first `omp for` section, with 2 for loops over the entire grid, we compute the 4 directions and store them in `moves_`.

Then, in another double for loop over the entire grid, we increase and decrease the values in `particles_` accord-

ingly. This is done serially, in a `omp single nowait` section as `particles_` is read a written as several locations at the same time.

Finally, the time is increased in a `omp single` section whose implicit barrier makes sure that the threads are synchronized before the next step. All of this in a `while` loop in a `omp parallel` section with the `stream` used by the set as `private` (it is initialized to a different value for each thread) and `r` where the random numbers are stored as `firstprivate` because it is initialized before.

We performed strong and weak scaling. The results (fig.14) are not as good as those of MPI. We can see still see the dip in speed up at 7 threads.

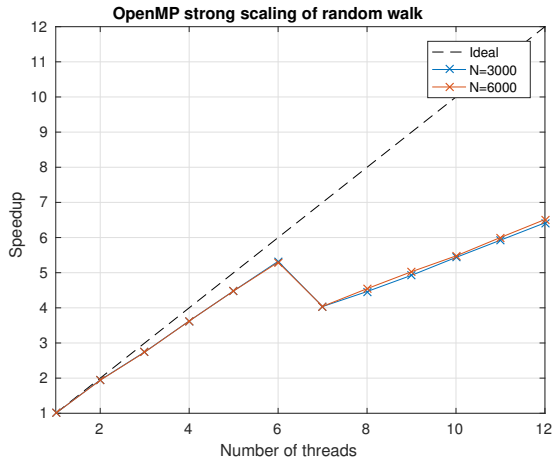

Fig. 14. Weak scaling of RW with OpenMP from 1 to 12 threads.

Same goes for the weak scaling (fig.15). We can note that it does not seem to depend on the grid size.

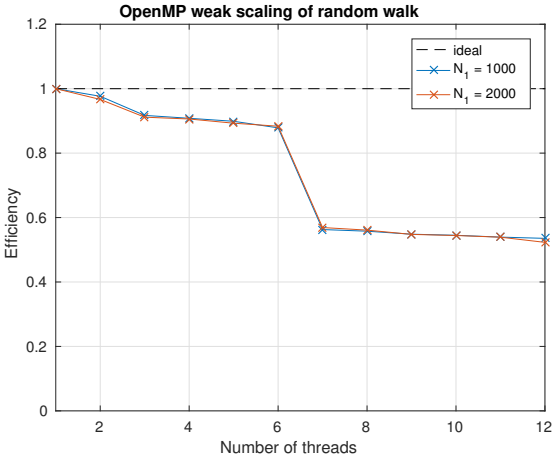

Fig. 15. Weak scaling of RW with OpenMP from 1 to 12 threads.

OpenMPI. We also parallelized the code both with OpenMP and MPI. It is mostly a mix of the 2 preceding methods, with the nuance that the communication part of MPI must be in `omp single` section so that they are done only once per process.

4. CONCLUSIONS

For the finite differences method with ADI, the best optimization method is to parallelize the code with OpenMP. Vectorization with AVX is not super effective, probably because most of it is already done by the compiler.

For the random walks, parallelizing with MPI was the most efficient method. The simple domain partition of this straightforward algorithm allowed a good implementation.