HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING PROJECT 2: DIFFUSION

Lorenzo Giacomel

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this project we implement and optimize two numerical methods for solving the diffusion equation. The former is the Alternating Direction Implicit method and the latter is the Random Walks method. For both methods we apply scalar optimization and parallelization on shared memory and on distributed memory. For the first we also used vectorization.

1. BACKGROUND

Diffusion processes can be modeled through the Partial Differential Equation (PDE)

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

Except for some cases (e.g., when the fundamental solution is known or with sinusoidal initial condition) the diffusion PDE cannot be solved analytically. For this reason it is of great interest to derive numerical methods to find approximated solutions.

Alternating Direction Implicit. A typical approach is to discretize the partial derivatives with finite differences. There are many different methods in this framework which, basically, differ for the formula used to discretize the derivatives. A requirement for a good method is usually to have at least second order of accuracy both in time and in space. Moreover there are methods that are not *unconditionally stable*, that means that they are unstable for some choices of the discretization steps in time and space. For example, some methods converge only when the CFL condition is satisfied, i.e. $\frac{D\delta t}{\delta x^2} < \frac{1}{2}$.

The Crank-Nicholson algorithm is a very popular choice because it is unconditionally stable and it has second order of accuracy both in time and space. This algorithm can become very slow when approximating a two dimensional diffusion process, since it requires the solution of a penta-diagonal linear system per time step (and epta-diagonal in three dimensions).

Teaming with Philipp Müller

To reduce the complexity of the linear system, it is possible to use the Alternating Direction Implicit (ADI) [1] method that performs separately one mono-dimensional step of the Crank-Nicholson method per direction. In two dimension the stencil is the following:

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

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

with

$$\frac{\partial^2 \rho_{i,j}^{*}}{\partial x^2} \approx \frac{\rho_{i-1,j}^{*} - 2\rho_{i,j}^{*} + \rho_{i+1,j}^{*}}{2\delta x^2},$$
$$\frac{\partial^2 \rho_{i,j}^{*}}{\partial y^2} \approx \frac{\rho_{i,j-1}^{*} - 2\rho_{i,j}^{*} + \rho_{i,j+1}^{*}}{2\delta y^2},$$

Now at each time step the solution can be computed solving two tridiagonal linear systems corresponding to the same matrix $\mathbf{A}$.

Random Walks. Random Walks (RW) method takes advantage of the fact that diffusion processes are closely related to Brownian Motion. A diffusion process is a Brownian Motion for an infinite number of particles. Now $\rho(\mathbf{r},t)$ corresponds to the density of particles at the point $\mathbf{r}$ at time t.

In this case an approximation of $\rho(\mathbf{r},t)$ can be achieved performing a Brownian motion for a high but finite number of particles. Moreover the position particles has to be constrained to a mesh of points $(x_0 + i\delta x, y_0 + j\delta y)$ with $i = 1 \dots N_x$ and $j = 1 \dots N_y$.

The random walks (RW) are performed moving a particle to one of the four neighboring cells with probability $\lambda = D\dfrac{\delta t}{\delta x^2}$.

This method is known to converge with order $\frac{1}{2}$ with respect to the number of particles in the sense that the standard deviation of the solution with respect to the theoretical solution is of order $-\frac{1}{2}$ with respect to the number of particles[2].

Test case. We test the methods on a square whose sides have unitary lengths with the boundary conditions

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

and with the initial density distribution

$$\rho(x, y, 0) = \sin(\pi x)\sin(\pi y),$$

for which the solution is given by

$$\rho(x, y, 0) = \sin(\pi x)\sin(\pi y)e^{-2D\pi^2 t}.$$

Experimental Setup. For the measurements we ran the codes on the Cray XC50 nodes of the Piz Daint supercomputer . The nodes are equipped with an with an Intel Xeon E5-2690 v3 CPU, with Haswell microarchitecture, with 12 cores which, according to Intel specifics, have a frequency of 2.60 GHz with Intel Turbo Boost turned off. The maximal bandwidth to memory is 68 GB/s. We compiled all ADI codes with g++ (GCC) 5.3.0 20151204 (Cray Inc.) and the flags -O3 -fopenmp -fno-tree-vectorize. We compiled all Random Walk codes with icpc (ICC) 17.0.1 20161005 and flags -mkl -O3 -fopenmp.

2. BASELINE IMPLEMENTATION

For both the considered methods we implemented a class called *Diffusion* containing an *advance* function, that computes one step of the algorithm, together with a function that writes the density into an external file, a routine that initialize the density at the first step and a function that computes the error with respect to the exact solution at a given time.

ADI. Besides all the other functions, for the ADI method we decided to create a function that pre-computes the LU decomposition of the $\mathbf{A}$ matrix. This pre-computation can be done because $\mathbf{A}$ is not time dependent and, thus, the coefficients for the Thomas algorithm can be computed only once before starting the time-loop. At every time steps the *advance* function is invoked, the current error in l^∞ norm is computed and the new error is calculated as the maximum between the old error and this new error.

In the *advance* function we first computed the partial time step in y direction and then the partial time step in x direction using for the Thomas algorithm using the precomputed coefficients.

In figure 1 it is possible to see the plots for the order of accuracy of the ADI method, which empirically confirm the theoretical second order both in time and space.

Moreover, we analyzed the position on the roofline model for this first baseline implementation. We computed the timing through the OpenMP timer. The roofline plot in figure 2 shows that the scalar implementation is compute-bounded,

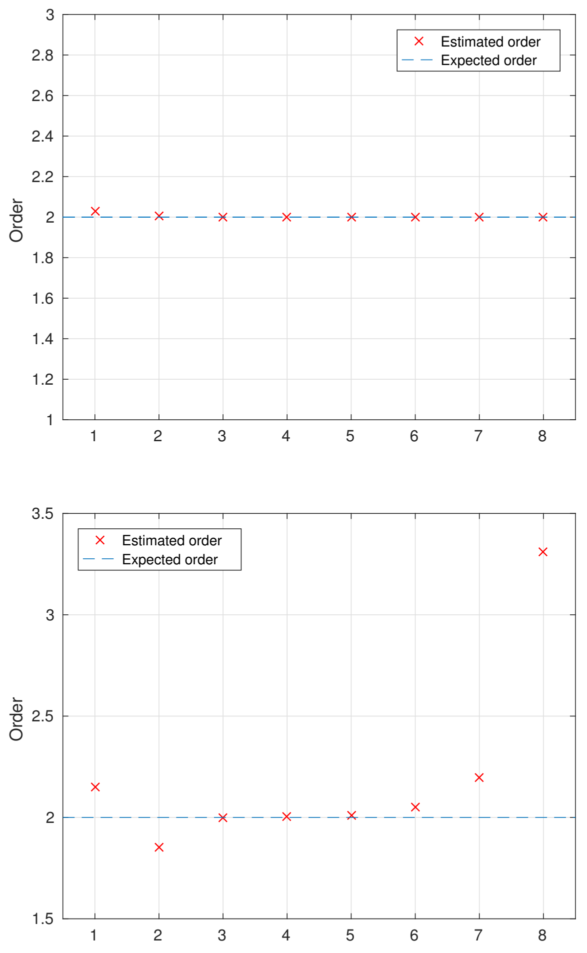

Fig. 1. Estimated accuracy order in space (upper) and time (lower) for ADI method.

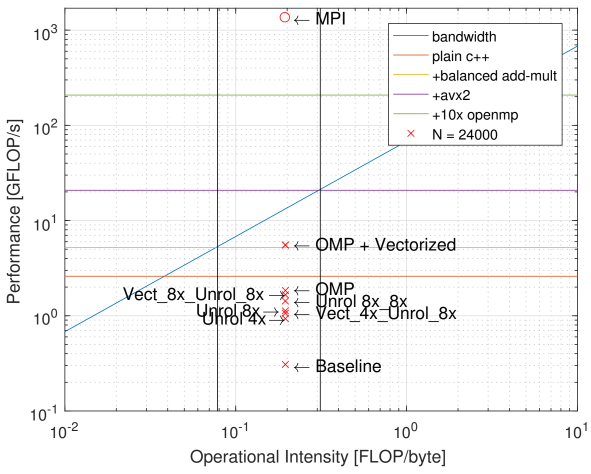

Fig. 2. Roofline model for the ADI method.

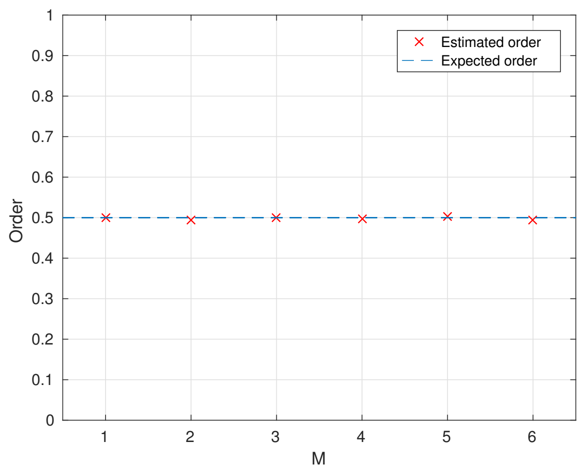

Fig. 3. Estimated order of accuracy with respect to the number of particles for RW method. ($M_0 = 10000000$, $N = 400$, $dt = 0.000001$)

but that it will be most likely memory-bounded when the vectorization will be enabled.

RW. For the random walk algorithm it is necessary to initialize the particles so that their density is proportional the actual initial density, this is done in the *initialize_density* function. At every time step the *advance* routine is invoked. In this function we looped over all the cells and for each of them we determined how many particles move in each direction by four binomially distributed random numbers. The binomially distributed random numbers are computed by the function *viRngBinomial* from the Intel MKL library. With this implementation it is possible that the number of moved particles is higher than the number of particles that were actually present on the node, we solved this issue repeating the random number generation every time that this situation happens.

Unfortunately, it was not possible to directly compute the operational intensity of this code because we did not have direct information about the operational intensity of the *viRngBinomial* function.

The code can be run specifying as input in the terminal the number of grid points per direction (N), the number of time steps (N_t) and the number of particles (M_0). In the code the time step (dt) is adjusted to have always $\lambda = 0.15$.

To validate the numerical solution we implemented the function *variance* that computes the variance of the solution at the final time. The plot in figure 3 shows that the convergence rate is $\frac{1}{2}$ accordingly to the theory.

3. OPTIMIZATIONS

3.1. Scalar Optimization

For scalar optimization we mostly used loop unrolling.

ADI. Looking at the roofline plot it is clear that our baseline implementation could be improved with some scalar optimization, since our code's performance is not close to the roofline. To improve the performance without using parallelization, we unrolled eight times both of the big loops of the two steps of the algorithm. The performance corresponding to this implementation is labelled on the roofline plot as *Unrol_8x_8x*. The optimization was effective as the performance is closer to the peak-performance line. Loop unrolling is beneficial in this case as it improves data-locality in the first loop, it enables better pipelining and it has less overheads in the for loops.

RW. When profiling the code for Random Walks method with *CrayPat*, it is possible to see that 70% of the time is spent in the *viRngBinomial* routine. For this reason optimizing the rest of the code, for example unrolling the external loop of the *advance* function does not affect positively the performance.

3.2. Vectorization

For vectorization the Haswell processors support AVX2 SIMD instructions. Thus we used the Intel intrinsics to perform vector operations in the registers.

ADI. We performed vectorization on the first for loop. We decided to vectorize only the first loop because according to our profiler around 75% of the total time is spent in this loop. The vectorization was repeated two times per loop iteration, reflecting the 8 times unrolling. As it can be seen in the roofline model we did not manage to have a good speedup, depending on the domain size we had in the worst case no speedup and in the best case 1.3 speedup. We also aligned the data and padded the rows of the matrix to use the intrinsics for aligned data but also this technique made no big difference. An hypothesis that could explain this behavior is that the first loop might be memory bounded. Comparing to the second half-step, the first has worse data locality and exploits less caching. For this reason more memory accesses are needed and it is more likely to hit the memory roofline.

We vectorized the second loop as well but did not achieve any speedup. The biggest issue with the second half step is that data are scattered in the memory so it would have been necessary to use some functions to gather and scatter them. We bypassed this problem by transposing the matrix in the last for loop of the first half step. Even though with this strategy it has been possible to vectorize the second half step, the memory accesses introduced for the matrix transposition make the first loop even more memory bounded.

Most likely for this reason we did not have any speedup applying vectorization to the second loop.

3.3. Shared-Memory Parallelization

For shared-memory parallelization we used OpenMP and we analyzed the scaling up to twelve cores.

ADI. We parallelized straightforwardly both the half steps using the OpenMp parallel for with schedule dynamic on the outer loops. The parallelization has been applied to the simple serial code and to the vectorized one. Moreover we tried to implement a parallelized version with touch-by-all initialization. Even though on NUMA architectures this approach can give benefits to code performance, we did not observe any improvement neither in the timings nor in the scaling. We measured strong scaling and weak scaling for different problem sizes and the results are reported in figure 4.

RW. Due to data dependencies in the final part of the main for loop, where the number of particles per cell is updated, it is not possible to parallelize the code as it is. For this reason we split the for loop into two for loops. In the first one the binomial random numbers are generated for each grid point and stored in a matrix. In the second one the number of particles on each grid point is updated. The first loop can now be parallelized, because there are no data dependencies. This approach is beneficial because the parallelized loop is the one containing the function *viRngBinomial*, which according to profiling results is the most time consuming. Weak scaling and strong scaling plots are shown in figure 5. In the strong scaling it is possible to see a plateau in the speedup for more than 6 threads. This probably happens because, when the first for loop is parallelized with many threads, the second loop becomes the most time consuming and the performance does not improve a lot anymore using a higher number of threads.

3.4. Distributed-Memory Parallelization

Parallelization on multi-node can be performed using MPI. For the ADI method we analyzed the scaling between 128 and 2048 nodes while for RW the analysis as been made between 1 and 24 nodes.

ADI. ADI needs special care for MPI parallelization because the first for loop iterates over rows and the second over columns. For this reason we decided to split the domain by horizontal stripes for the first half-step and by vertical stripes for the second half-step. This means that, after that the intermediate density is computed, each node communicates with all the others to change the domain division. MPI provides an useful routine for this process, which is *MPI_Alltoall*. Using this routine two times it is possible to pass from the horizontal division to the vertical one, and to switch back after the second half step. In order to

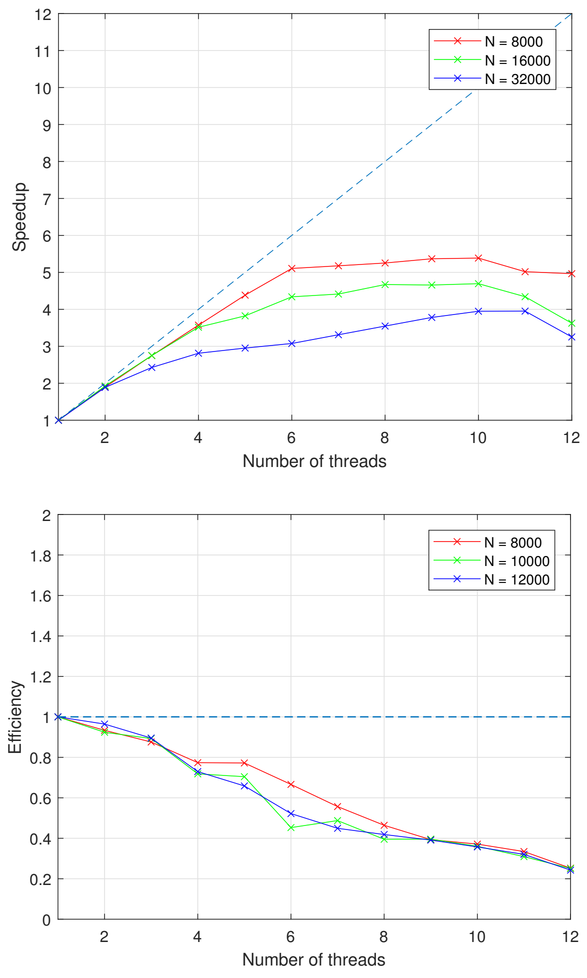

Fig. 4. Upper plot: OpenMP strong scaling for ADI method for N grid points per direction. Lower plot: OpenMP weak scaling for ADI method starting with N grid points per direction.

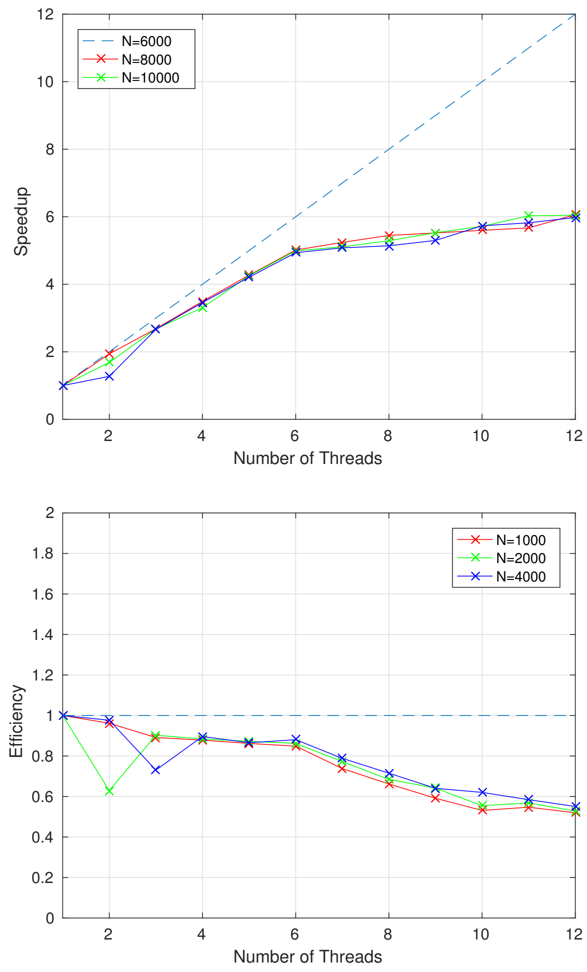

Fig. 5. Upper plot: OpenMP strong scaling for RW method for N grid points per direction. Lower plot: OpenMP weak scaling for RW method starting with N grid points per direction.

use *MPI_Alltoall*, we created a datatype to send blocks of the matrix and one to receive them. Moreover, we used ghost cells to communicate the density value at the boundary of the stripes. To simplify the communications, we used a Cartesian topology. We used asynchronous communication for the ghost cells and the latency for the communication is hidden with the computation of the density on internal nodes. We applied distributed-memory parallelization to both the serial and vectorized codes with similar results. Overall this approach guarantees a reasonable speedup. The plots for strong and weak scaling between 128 and 2048 nodes are shown in figure 6. It can be observed that for big matrix sizes the scaling overcomes the theoretical speedup. It can be inferred that the reason is that the size of single communications is bigger for smaller number of processors, i.e. dividing the domain with less processors it is necessary to send and receive bigger blocks. Under the hypothesis that *MPI_Alltoall* is more time consuming for smaller number processors, it seems reasonable that the speedup is higher than expected for higher number of processors.

RW. Random Walks method can be parallelized as it is with MPI. We simply split the domain into horizontal stripes assigning each stripe to a different node. We used rows of ghost cells to communicate to the upper and lower stripe how many particles move upwards and downwards. This communication is performed asynchronously with *Isend* and *Irecv*, but it is followed by an *MPI_Waitall* so it's actually done synchronously. The size of communication is relatively small, while the external for loop is fully parallelized, so this parallelization scheme scales particularly well. The resulting strong and weak scaling plots can be found in figure 7.

4. CONCLUSIONS

We conclude summarizing the results achieved with our optimizations.

ADI. Among all the optimization techniques we tried for ADI the one that stands out is MPI parallelization. The reason is that our code for ADI is memory bounded and the best improvements are achieved when the maximum memory bandwidth is increased, which is the case of distributed memory parallelization.

RW. Also for RW the best results are obtained when performing distributed memory parallelization. This is because comparing with shared memory parallelization we were able to parallelize a bigger portion of the code introducing a small amount of communications.

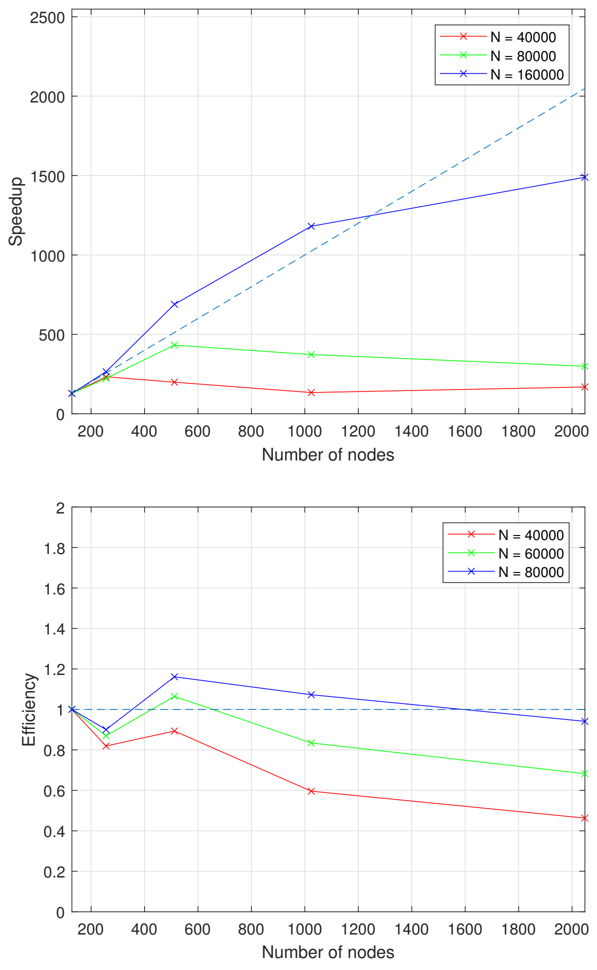

Fig. 6. Upper plot: MPI strong scaling for ADI method for N grid points per direction. Lower plot: MPI weak scaling for ADI method starting with N grid points per direction.

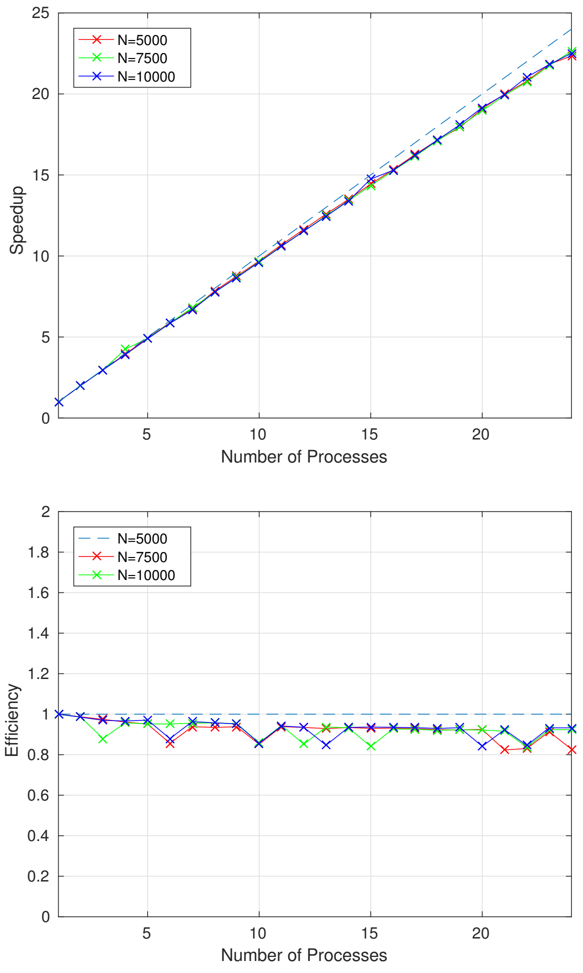

Fig. 7. Upper plot: MPI strong scaling for RW method for N grid points per direction. Lower plot: MPI weak scaling for RW method starting with N grid points per direction.

5. REFERENCES

[1] D. W. PEACEMAN and JR H. H. RACHFORD, *The Numerical Solution of Parabolic and Elliptic Differential Equations*, SIAM, 1955.

[2] SANDRO SALSA, *Partial Differential Equations in Action*, chapter 2, pp. 43–62, Springer, 2008.