OPTIMIZATION AND PARALLELIZATION OF DIFFUSION SOLVERS USING ALTERNATING DIRECTION IMPLICIT AND RANDOM WALK METHODS

Samuel Maloney

Department of Mathematics
ETH Zürich
Zürich, Switzerland

ABSTRACT

Optimized and parallelized code for numerical simulation of the diffusion equation is presented. Two different algorithms, Alternating Direction Implicit (ADI) and Random Walk (RW) are studied. The algorithms are implemented in C++ and verified against an analytic solution, and then scalar optimizations, AVX vectorization, and OpenMP parallelization are carried out. Benchmarking tests are conducted on the Piz Daint cluster to determine single-core performance and runtime improvements, as well as strong and weak scaling characteristics on multiple cores of a single node. Approximate performance gain of 5X for ADI, and runtime speedups of approximately 6X and 3000X for ADI and RW respectively are observed, along with acceptable OpenMP scaling.

1. BACKGROUND

Motivation. Diffusion is the process by which a quantity of interest spreads from regions of high density to regions of low density and are integral in the study of fluids. Diffusion terms arise in all numerical models of computational fluid dynamics, which are used for such problems as aerodynamic design, turbine flows, chemical reaction mixing, and many others. Efficient and accurate simulation codes for diffusion are thus of great importance in many fields.

In this study, two different algorithms to simulate the diffusion process, Alternating Direction Implicit (ADI) and Random Walk (RW), were implemented, optimized, and parallelized. The mathematical formulations for diffusion in general and for each of these two methods specifically are presented next.

Mathematical Formulation. The process of diffusion is driven by it's conentration gradient according to the diffusion equation:

$$\frac{\partial \rho(\mathbf{r},t)}{\partial t} = D\Delta\rho(\mathbf{r},t) \tag{1}$$

where $\rho(\mathbf{r},t)$, with position $\mathbf{r}$ at time t, is the quantity of interest and D is the diffusion constant and is a given value for the system. For this study, homogeneous Dirichlet boundary conditions are used at all boundaries. The initial density distribution used for this study is:

$$\rho(x,y,0) = \sin(\pi x)\sin(\pi y) \tag{2}$$

and it's corresponding analytic solution, which was used for verification, is:

$$\rho(x,y,t) = \sin(\pi x)\sin(\pi y)e^{-2D\pi^2 t} \tag{3}$$

The studied simulation domain is a unit square with x and y each ranging from 0 to 1, which is discretized using a uniform grid such that $\delta h = \delta x = \delta y$. A superscript n is used to denote the timestep $t_n = n\delta t$ and subscripts i and j to denote the indices of the nodes in the mesh $x_i = i\delta x$ and $y_j = j\delta y$ such that $\rho^n_{i,j} = \rho(x_i, y_j, t_n)$.

Alternating Direction Implicit. The ADI method is an operator splitting scheme based on finite difference discretizations of the derivatives. It works by splitting the each time step into two half tiem steps, in the first of which the time integration is carried out implicitly in the x-direction and explicitly in the y-direction, and in the second half time step the directions are switched such that the integration in the x-direction is explicit and in the y-direction is implicit.

Using a second order centered approximation for the spatial derivative in Eq. (1) means that each implicit integration step requires only the solution of a tridiagonal linear system, for which the Thomas Algorithm (for formulation see [1]) can be used to efficiently compute the result. These tridiagonal systems can be found by considering the discretization equations:

$$\rho^{n+\frac{1}{2}}_{i,j} = \rho^n_{i,j} + \frac{D\delta t}{d\delta x^2}(\rho^{n+\frac{1}{2}}_{i-1,j} - 2\rho^{n+\frac{1}{2}}_{i,j} + \rho^{n+\frac{1}{2}}_{i+1,j}) \tag{4}$$

$$\rho^{n+1}_{i,j} = \rho^{n+\frac{1}{2}}_{i,j} + \frac{D\delta t}{d\delta x^2}(\rho^{n+1}_{i,j-1} - 2\rho^{n+1}_{i,j} + \rho^{n+1}_{i,j+1}) \tag{5}$$

These can be rewritten in the form $\mathbf{A}\rho^* = \rho^{*-\frac{1}{2}}$ where $\mathbf{A}$ is the tridiagonal matrix to be solved at each half timestep.

Random Walk. Diffusion can also be simulated by tracking the random motion of a set of M identical particles with no inertia. A probability $\lambda = D\delta t/\delta x^2$ is defined, such that at each time step, each particle has a $1 - 4\lambda$ probability of remaining in the same position. If it does not stay, it is then moved to one of the four neighbouring nodes, with an equal probability of moving in any direction. Clearly, one requires $lambda < 1/2$ to maintain a positive probability of staying.

The initial conidition for the numer of particles at each grid point $m_{i,j}$ is given as:

$$m_{i,j}^0 = \left\lfloor \frac{M\rho(x_i, y_j, 0)}{\iint_\Omega \rho(x, y, 0)\,\mathrm{d}x\,\mathrm{d}y} \right\rfloor \qquad (6)$$

And the approximate solution after a given timestep can then be calculated as:

$$\rho_{i,j}^n = \frac{m_{i,j}^n}{M} \iint_\Omega \rho(x, y, 0)\,\mathrm{d}x\,\mathrm{d}y \qquad (7)$$

2. BASELINE IMPLEMENTATION

In this section, the basic implementation and verification of the described algorithms is presented. All code is based on the Diffusion2D class structure of the solution code provided with the course exercises. All code for each version of the class is contained within a single .cpp source file, and includes user defined header files *timer.hpp* and *tsc_x86.hpp* for measuring runtime in seconds and cycles respectively. The *timer.hpp* file is from the HPCSE I course exercises, and the *tsc_x86.hpp* file is from the exercises of the How to Write Fast Numerical Code course exercises.

The programs take arguments to set the diffusion constant D, size N of one dimension of the grid, timestep δt, number of repititions n_{runs} to run the simulation, and the number of timesteps n_{steps} to run each simulation for. The RW codes also take a value for the numer of particles M in the simulation. For all codes the values for n_{runs} and n_{steps} are optional, and default to $n_{runs} = 1$ and $n_{steps} = 0.1/\delta t$ respectively. The ordering of these input arguments can be displayed by running the program with no arguments.

For each simulation run, a new Diffusion2D class object is initialized, with one-time initialization routines happening outside of the timing structures. Once initialization is complete, the simulation is is run to completion by a single call to the `run_simulation` function, and this call is the only part of the program which is timed and contains all code which will be optimized.

ADI. An order verification study (OVS) of the ADI code in both spatial and temporal diensions was performed by comparing the simulation results to the analytic solution in Eq. (3). The results can be seen in Fig. 1 and show the implementation to be second order accurate in both space and time, as expected for this discretization scheme.

The ADI algorithm complexity should be $O(N^2)$ as it does a constant number of operations for updating value at each point in the grid. It's runtime scaling (see 4.1, Fig. 8) is also found to agree with this expected complexity.

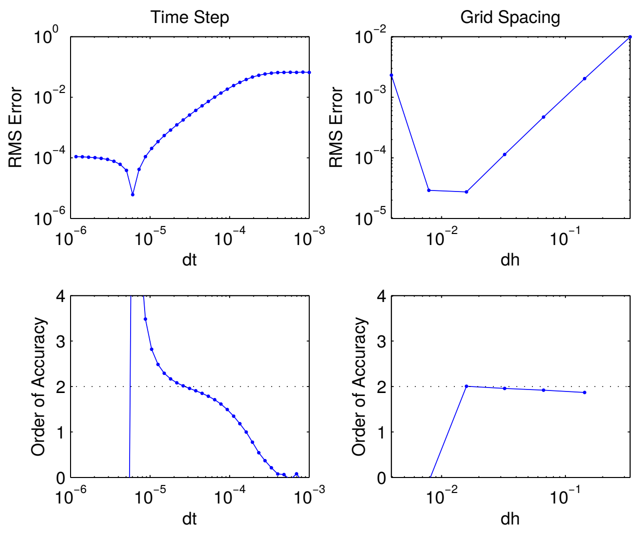

Fig. 1. Order verification study of the ADI code in space and time.

For performance analysis, the cost of the ADI code is defined as the number of floating point operations (FLOPs) including additions, multiplications, and, for the baseline implementation only, divisions. Moreover, the data movement in bytes between various levels of cache was measured using the Hardware Performance Counter (HWPC) functionality in CrayPat to quantify the operational intensity.

The results of a roofline analysis are shown in Fig. 2 for the baseline code. Three data series are presented, which use the same performance data but compute the operational intensity with respect to different levels of the cache heirarchy. The imlementation appears compute bound with respect to memory tranfers with RAM, but for large N it becomes clearly memory bound at higher levels of the cache, albeit with still siginificant room to improve towards the memory bandwidth rooflines. Maximum performance of 0.734 FLOPs/cycle, or 18% of peak scalar performance is observed. On the test machine, with an Intel Haswell central processing unit (CPU), the peak performance is calculated by considering the maximum throughput of 2 fused multiply add (FMA) instructions per cycle, giving 4 total FLOPs per cycle. Data for the cache bandwidths used in this (and subsequent) roofline plots was obtained from [3] and [4].

RW. One of the most important aspects of the RW implementation is the choice of pseudorandom rumber generator (PRNG) used. Knowing that parallelization would eventually be necessary, the SITMO PRNG [5] was initially selected. It claims to be faster than the C++ Mersenne Twister, passes all BigCrush test in the TestU01 framework, and provide easy access to many non-overlapping streams

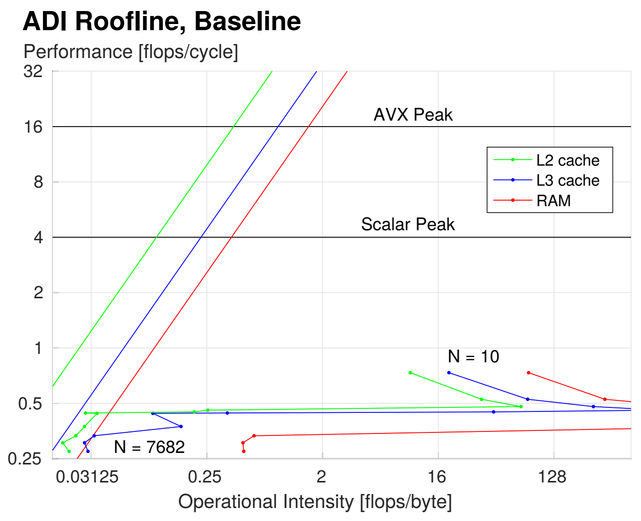

Fig. 2. Roofline analysis of the ADI baseline code.

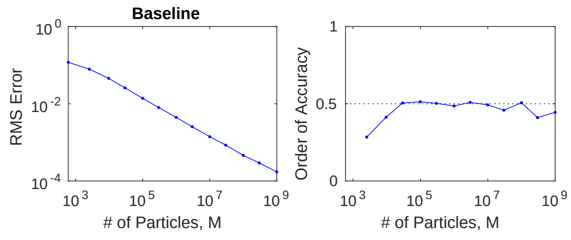

Fig. 3. Order verification study of the baseline RW code for the number of simulation particles.

which would be useful for later parallelization. For the baseline implementation only, the code loops over all nodes and then at each node loops over all particles at that node. For each particle a random number is generated to determine if it stays or in which direction it should move.

An OVS for the baseline RW code is shown in Fig. 3 with respect to the number of particles in the simulation, again comparing to the analytic solution. This study confirms the baseline to converge as $\sqrt{M}$ as expected from theory. As explained in section 3.2, algorithmic changes were made during later optimizations of the RW code, and so additional OVSs were carried out on these later versions to ensure they remained correct. Results for these verifications are shown in Fig. 4 and confirm that the $\sqrt{M}$ convergence is preserved.

Scaling for the RW method depends on which version is being considered. The baseline implementation should scale as $O(M)$, where M is the number of particles in the simulation. This is because a random number must be generated for each particle in the simulation to determine it's movement for a given timestep, independent of how many grid points there are in the simulation. The later versions

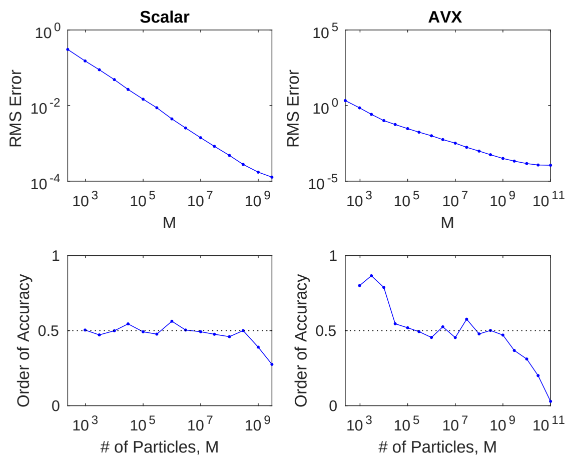

Fig. 4. Order verification studies of the optimized RW codes for the number of simulation particles.

that use a binomial random variable (see 3.2) to simulate particle movement scale as $O(N^2)$ because exactly four such random variables must be evaluated for each grid point, independent of the number of particles in the simulation.

Measurements of this runtime scaling (see 4.2, Fig. 11) confirm $O(N^2)$ for all versions of the code. This is as expected for the optimized versions, but also matches for the baseline version, since the number of particles was chosen as $M = 1000N^2$, meaning the $O(M)$ complexity is equivalent to $O(N^2)$ complexity for these experiments.

3. OPTIMIZATION METHODS

In this section the main optimizations that were undertaken are explained. Optimizations of the ADI and RW methods are discussed in separate subsections. For each algorithm, the optimizations are broken down into three revisions: the *Scalar* revision contains only optimizations that do not involve manual vecotorization or parallelization (although it is noted that the compiler was not prevented from vectorizing, so this naming scheme is not meant to imply that the machine code is not vectorized, only that it was not manually attempted at this stage); the *AVX* revision then adds manual vectorization using intrinsics; and the *OpenMP* revision add parallelization using OpenMP to run on a single full XC50 node of Piz Daint.

3.1. ADI

Scalar. Several different types of optimizations were included in this revision. Firstly, the number of operations required was greatly reduced by precomputing several factors needed for the Thomas algorithm that were the same for each row/column and timestep. This required an additional two vectors of length N to store the precomputed values, but greatly reduced the FLOP count overall, and in particular removed all division operations from the simulation loop.

Secondly, unrolling of the outer loops was performed to take advantage of the spatial locality in the algorithm. It achieves a speedup because updating the values for each row/column requires the values from the immediately adjacent rows/columns, so by unrolling the loop we expose independent operations in adjacent rows which use the same data, thus decreasing the data movement required and increasing Instruction Level Parallelism (ILP). It was decided to unroll by a multiple of four, in preparation for vectorization with AVX, and unrolling factors of four, eight, and twelve were tested, with a factor of eight determined to be optimal on the test system.

Lastly, Single Static Assignment (SSA) was introduced throughout the code to promote optimal register allocation by the compiler for all reused data values and computations.

AVX. Manual vectorization of the code with AVX intrinsics was carried out in this revision. In the final version of this code, only the loop over the columns was manually vectorized, as nothing that was attempted to the row loop was able to reduce the runtime. This is mostly due to the row-major storage order of C++ arrays, which mean that data for adjacent columns is nicely aligned for loading to and from AVX vectors, but the data from adjacent rows requires some sort of gathering and scattering operations to fill the vectors with the correct data. Several methods of achieving this were tested, but were found ot be slower than simply letting the compiler handle the scalar code.

The first method attempted involved storing the half-timestep result in transposed form, such that both loops were loading their data column-wise, thus requiring only the single scatter operation for each vector of values updated, abeit for both loops, rather than the one scatter and three gather operations that would have been required to read and store in the column-wise loop. The second method attempted involved unrolling the inner loop by a factor of four, such that the full vectors of data from each load operation could be used, rather than just the single value used from each load operation used in a gather. This reduced the number of load instructions required by a factor of four, but does replace them with a number of shuffle and permute operations to arrange all of the loaded data properly into vectors for the computations (AVX transposition code based on [6]).

One last thing that was attempted at this stage was unrolling of the vectorized inner loop in an attempt to improve piplining of operations. An unrolling factor of three was selected as it was the largest factor for which the number of required AVX vectors in the code did not exceed the number of available AVX registers on the CPU. In this way, load instructions for data for future iterations were interleaved with computations for current iterations, but unfortunately no improvement in runtime was attained, and so this unrolling of the inner loop was not included in the final code for this revision.

Implementations for the required scatter, gather, and rearranging operations were hand coded and optimized using throughput, latency, and execution port data from Agner Fog's tables [7], which were also used for analysis of instruction level dependencies in the attempts to improve piplining.

OpenMP. Initial parallelization of the ADI code with OpenMP was quite straightforward. The code was restructured slightly, such that instead of a loop in the main function calling an `advance` function for each iteration, the loop was placed directly around the simulation code inside of a `run_simulation` function, as mentioned in section 2. This allowed for a single `omp parallel` region to be created at the beginning of the simulation that would only be exited upon completion, preventing joining and spawning of threads between iterations and removing OpenMP overhead from within the loop. After that, simple parallelization was achieved with `omp for` on both loops, using a `no wait` clause on the second loop because synchronization would already be achieved by the `omp single` used for updating the step counter only once at the end of each iteration.

However, one major issue which remained with this initial implementation was false sharing. Although the individual threads never need to write to exactly the same entries of the array, they are often writing to nearby entries at the boundaries of their working set. This means that the boundaries of each working set should be cache line aligned in order to prevent false sharing between threads. To attain this, a check was added to the code requiring that the number of grid points N be a multiple of eight, since there are eight doubles in a 64 B cache line. An `aligned_alloc` was then used to allocate 64 B aligned memory for storage, and as the outer loops were already unrolled by a factor of eight this ensured that each thread's working set did not share cache lines with its neighbours.

3.2. RW

Scalar. The main optimization of this version lies in no longer looping over every particle and indivdually computing a random number to determine its potential movement, but rather looping only over each node and generating four binomially distributed random numbers to determine the ensemble number of particles that move in each direction. This makes the runtime scaling independent of the number of particles in the simulation, allowing much higher accuracy simulations without prohibitively increasing the runtime. It is noted that because the sum of the four binomially distributed numbers can be greater than the actual number of particles on the node, particularly for small M, a check was added to the code to handle negative nodal values during simulation, but they can still show up as negative values in the final density solution.

A further optimization involved moving the computation out of the simulation loop and into it's own function, such that the conversion was not calculated on each time step, but could be manually called after completion of the simulation to compute the final approximate solution.

AVX. Vectorization of the code using manual Single Instruction Multiple Data (SIMD) programming was done by leveraging Intel fused-multiply-add (FMA) and advanced vector extension (AVX) intrinsics. However, it was quickly discovered that changing the code to use AVX intrinsics had only a minimal speedup because the PRNG itself was where most of the runtime was spent, and so vectorized PRNG options were investigated. Intel's Math Kernal Library (MKL) Vector Statistics (VS) package implements a Streaming SIMD Extension (SSE) version of its fast mersenne twister PRNG [8], and so this was integrated with the AVX intrinsics to create a fully vectorized version of the code.

Because the RW algorithm updates the four neighbouring nodes for any given one being looped over, it was found to be faster if only six nodes were looped over at a time, instead of the maximum eight which would correspond to a full AVX vector of 32-bit integers. This allowed the values being added to the neighbouring nodes to be cleverly aligned within the AVX vectors and added using basic ADD intrinsics, one on top of the other, and eliminated a small dependency chain of loads and stores that occurred when the loop was originally unrolled by a full factor of eight.

OpenMP. Again because of the RW algorithm updating the nodes neighbouring the ones being looped over, the main challenge for OpenMP parallelization was preventing race conditions at the boundaries of each thread's working set. To this end, an array of `omp_mutex` objects (from the C++ class implementation provided in HPCSE I) was introduced, with each lock in the array matching to one of the working set boundaries. Working set indices for each thread were manually computed and the inner for loop was separated into three parts, one each for the first two and final two rows in each working set, which are the ones that have overlap with neighbouring threads, and one for the interior cells that do not have any possible race condition. Each thread then must acquire a lock before entering one of its boundary loops, with the lock for the first rows loop of one thread corresponding to the lock for the final rows of the previous thread to prevent race conditions.

Each thread also requires independently generated random numbers, and so each has its own PRNG stream which is made to be `threadprivate`. The streams are ensured to be non-overlapping by using the skip ahead functionality provided by the MKL VS library, with the number of entries in the sequence to be skipped being calulated from the number of grid points and required timesteps.

4. OPTIMIZATION RESULTS

In this section the results of numerical benchmarks of the code at various stages are presented. Runtime and scaling experiments were performed for both algorithms, as well as performance and roofline analyses for ADI.

Experimental setup. All data was collected using a single XC50 node of Piz Daint, which has a 12 core Intel Xeon E5-2690 v3 running at 2.6 GHz [2]. Each physical processor core has a 64 KB L1 cache split equally into two 32 KB data and instruction caches, and a 256 KB unified L2 cache [9]. The 30 MB L3 cache is shared between all cores [10], and there is 24.9 GB/s bandwidth to main memory [3]. The Intel C++ Compiler v17.0.1 was used with "-O3 -std=c++11 -DNDEBUG -march=core-avx2 -fno-alias -qopenmp -mkl" flags. The Intel time stamp counter (TSC) was used for runtime measurements and CrayPat was used for cache miss measurements.

The inputs are the number of grid points in domain discretization, the timestep between iterations, and the number of timesteps computed. We used grids of various sizes, up to $N \times N = 7680 \times 7680$. The diffusion constant $D = 1$ is used for all simulations, and for RW simulations $M = 1000N^2$ particles was used. For ADI, 1000 timesteps were computed, and for RW 50 timesteps were computed to achieve a reasonable runtime for measurements. Each simulation was repeated 10 times, with the minimum cycle count used for the runtime and the average cache misses used for the memory transfer (the average is used only because it would have prohibitively time-consuming to measure minimum values with CrayPat).

All experiments were performed with warm cache. This is largely to simulate real world usage scenarios, where the simulation would most likely be performed immediately after initialization while the data is still potentially in cache, exactly as in these experiments. It is noted, however, that the initial cache state should have little effect on the results in either case, as the effects will be amortized over the number of iterations performed, and should become negligible for the relatively large number of iterations used.

4.1. ADI

Roofline, runtime, and scaling experiments were conducted on the ADI code.

Results. The runtime plots of various revisions of the ADI code are shown in Fig. 8. Each consecutive revision of the code resulted in a speedup.

4.2. RW

Benchmarking experiments were conducted on the RW code

Results. Only runtime and scaling analyses of the RW code were conducted and are presented here. Performance

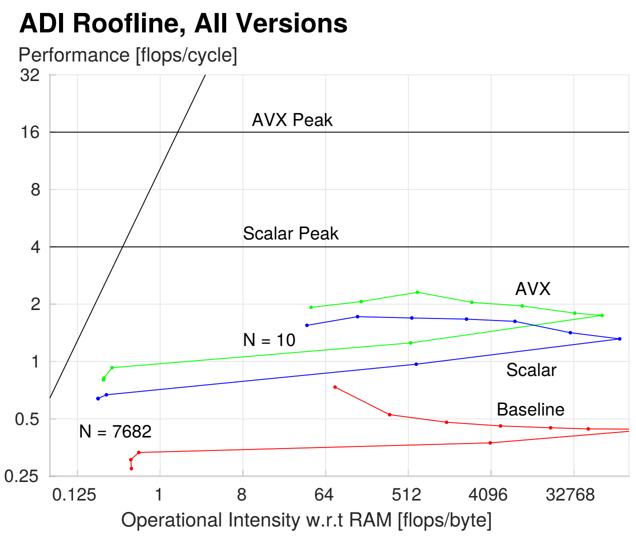

Fig. 5. Roofline analysis of the ADI code.

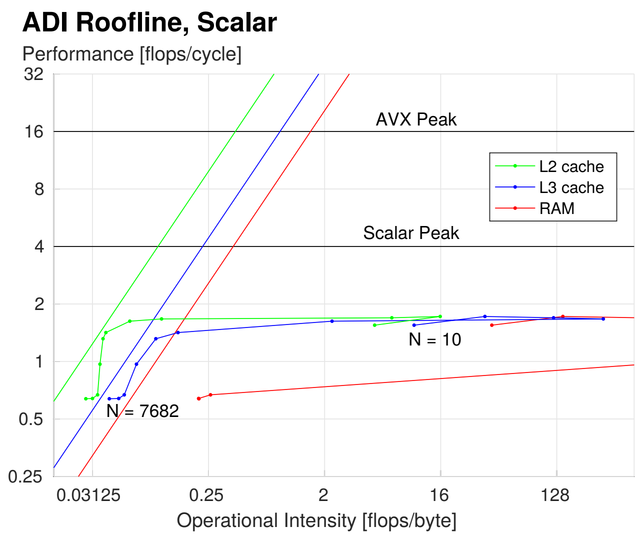

Fig. 7. Roofline analysis of the ADI scalar code.

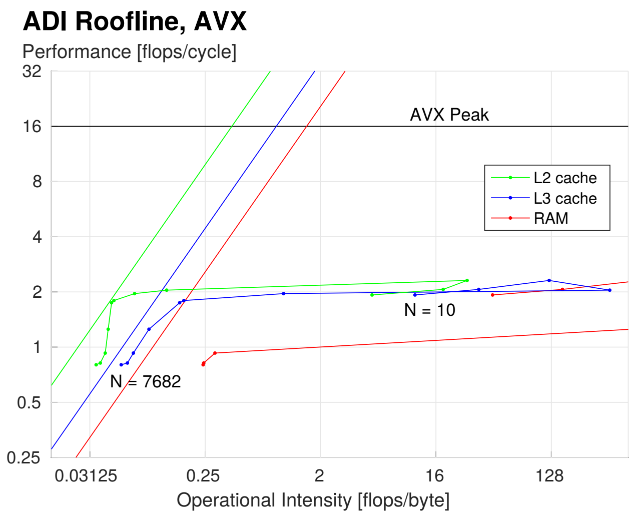

Fig. 6. Roofline analysis of the ADI AVX code.

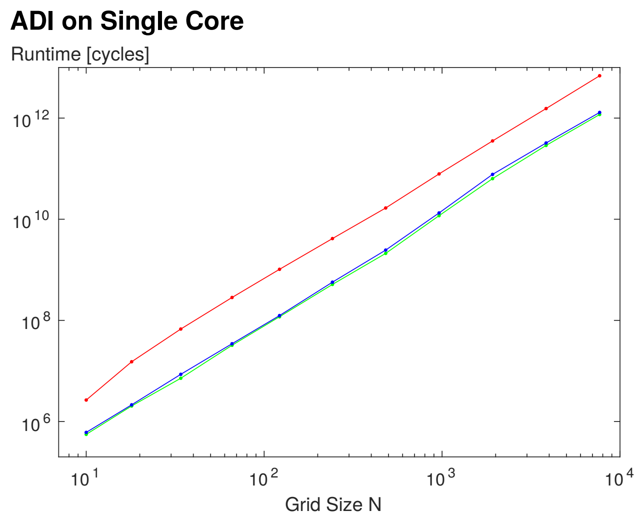

Fig. 8. Runtime analysis of the ADI code.

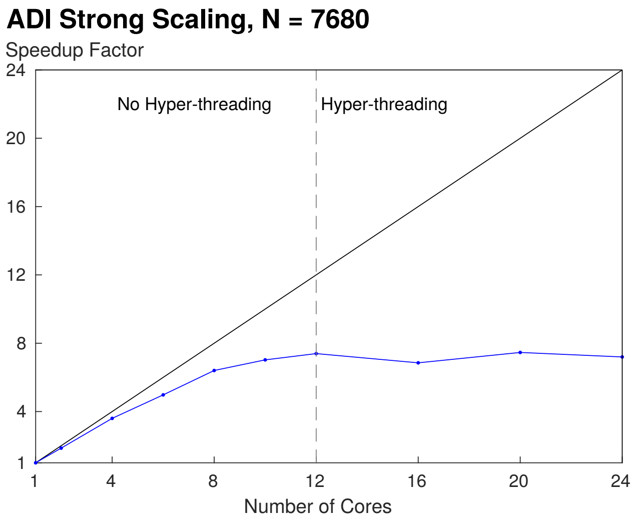

Fig. 9. Strong Scaling analysis of the ADI OpenMP code.

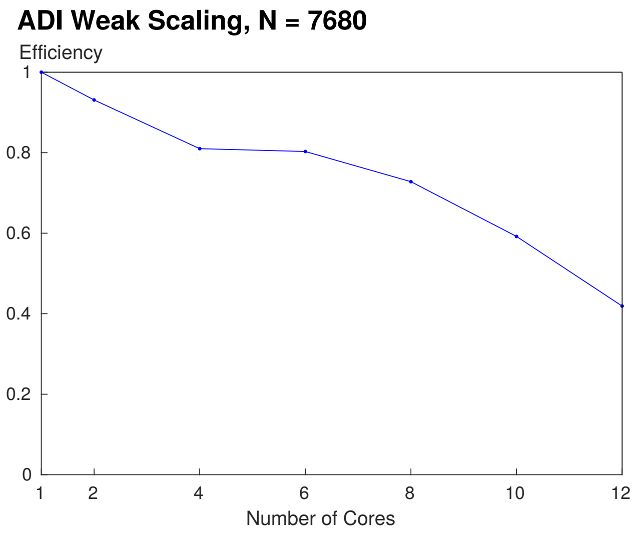

Fig. 10. Weak Scaling analysis of the ADI OpenMP code.

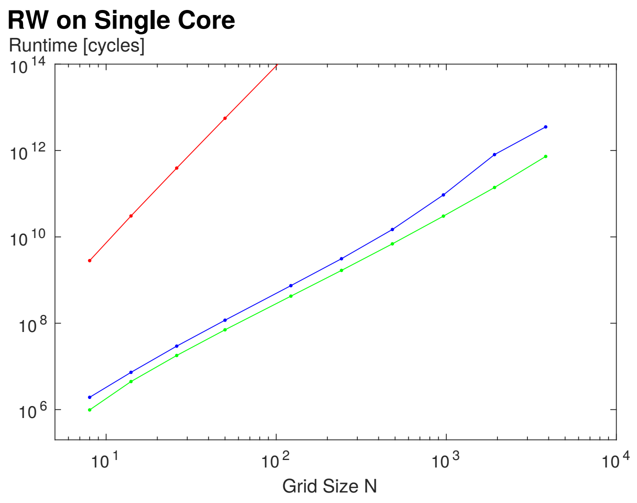

Fig. 11. Runtime analysis of the RW code.

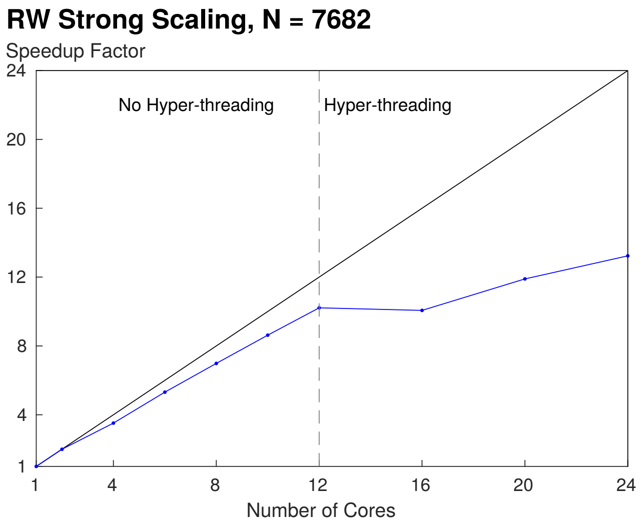

Fig. 12. Strong Scaling analysis of the RW OpenMP code.

and roofline analyses were not carried out because of the ”black box” nature of the PRNGs in the code, which render it impossible to measure or even usefully approximate the number and type of operations which occur during the simulation.

5. CONCLUSION

Conclude things

6. REFERENCES

[1] Wikipedia. (2017, May 16). *Tridiagonal matrix algorithm* [Online]. Available: https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

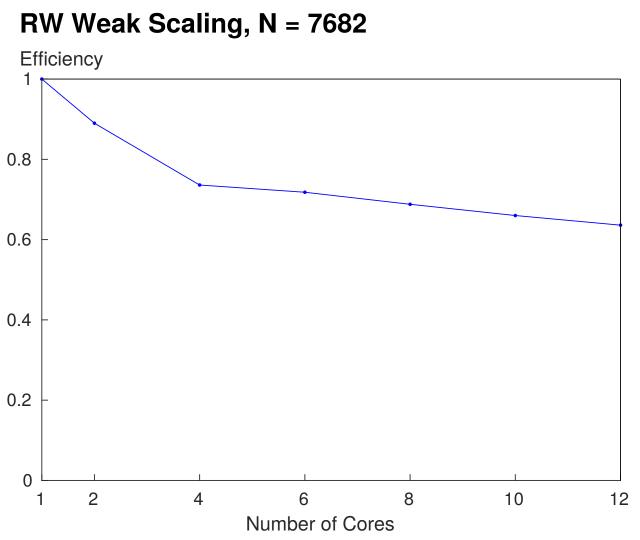

Fig. 13. Weak Scaling analysis of the RW OpenMP code.

[2] CSCS. (Accessed 2017, Aug). *Piz Daint* [Online]. Available: http://www.cscs.ch/computers/piz_daint/

[3] 7-CPU. (Accessed 2017, Aug). *Intel Haswell* [Online]. Available: http://www.7-cpu.com/cpu/Haswell.html

[4] Intel Corporation. (2016, Jun). *Intel 64 and IA-32 Architectures Optimization Reference Manual* [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

[5] T. van den Berg. (Accessed 2017, Aug). *High Quality C++ Parallel Random Number Generator* [Online]. Available: https://www.sitmo.com/?p=1206

[6] user2927848. (2016, Mar 23). *m256d TRANSPOSE4 Equivalent* [Online]. Available: https://stackoverflow.com/questions/36167517/m256d-transpose4-equivalent

[7] A Fog. (2017, May 2). *Instruction tables* [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[8] Intel Corporation. (Accessed 2017, Aug). *SFMT19937* [Online]. Available: https://software.intel.com/en-us/node/590406

[9] CPU-World. (Accessed 2017, Aug). *Intel Xeon E5-2690 v3 specifications* [Online]. Available: http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2690%20v3.html

[10] Intel Corporation. (Accessed 2017, Aug). *Intel Xeon Processor E5-2690 v3* [Online]. Available: http://ark.intel.com/products/81713/Intel-Xeon-Processor-E5-2690-v3-30M-Cache-2_60-GHzl