AMSC project report: multigrid

Mei Matteo

September 6, 2024

1 Introduction to the problem

The Poisson equation is a common PDE with applications in mechanics and electromagnetism. In it's simplest one dimensional form it has the structure:

$$\begin{cases} -u''(x) = f(x) \quad x \in (0, L) \\ u(0) = \alpha \\ u(L) = \beta \end{cases}$$

The report expands the finite difference method as a suitable computational method to solve second order linear problems with Dirichlet boundary conditions.

1.1 Finite differences scheme

The goal of a PDE discretization is approximating a continuous differential operator with a computable quantity. Finite differences methods use Taylor's expansion to produce formulas of arbitrarily accurate order. For the Poisson problem it's common to use a second order accurate formula:

$$u''(x) = \frac{-u(x-h) + 2u(x) - u(x+h)}{h^2} + o(h^2)$$

That requires the domain problem Ω to be approximated by a square lattice $\{x_0 = 0, x_1, \dots, x_n = L\}$ to yield a system of linear equations

$$\begin{cases} u(x_0) = \alpha \\ -u(x_{i-1}) + 2u(x_i) - u(x_{i+1}) = h^2 f(x_i) \\ u(x_n) = \beta \end{cases}$$

Given the particular structure of the operator, there exist a unique solution to this linear problem and since this is a second order formula, the error of the discretization is $O(h^2)$. The convergence order will be a metric of the quality of the program implementation

1.2 2D specialization

$$\begin{cases} -\Delta u = f(x,y) & (x,y) \in \Omega \\ u(x,y) = g(x,y) & (x,y) \in \partial\Omega \end{cases}$$

The bidimensional Poisson problem is discretized with the following formula:

$$\Delta u(x,y) = \frac{-u(x-h,y) - u(x+h,y) - u(x,y-h) - u(x,y+h) + 4u(x,y)}{h^2} + o(h^2)$$

This yields a sparse linear system but, contrary to its one dimensional counterpart, this system is much more demanding from a memory perspective to direct solvers because of the fill-in phenomenon.

1.3 Linear problem landscape

We have outlined how finite differences discretization of the Poisson problem leads to solving non-singular linear systems of the form:

$$Ax = b$$

It is sufficient to say that when node count increases (and thus the discretized solution gets closer to the exact one) the conditioning of the linear problems increases quadratically with the number of nodes. This affects the convergence speed of classic "black-box" iterative methods and, on the other hand, direct solvers suffer from the the aforementioned fill-in.

2 Multigrid

The geometric multigrid method is an iterative method that takes the best from direct and iterative solvers. The core idea of the method is the coarse grid correction, a procedure for which the error equation

$$Ae = r$$

is solved on a coarser grid, typically a grid with double the step. The coarse error is then interpolated and used as a correction of the fine grid solution and the process can be repeated until the desired convergence condition is met. This is called the two level multigrid but the coarse correction idea can be extended recursively to produce a hierarchies of grids, each one solving the error equation of the upper level (multilevel multigrid). The building blocks of the method are:

- **Relaxing:** it means to apply a smoother (Gauss Seidel or Jacobi method) on the current grid solution;

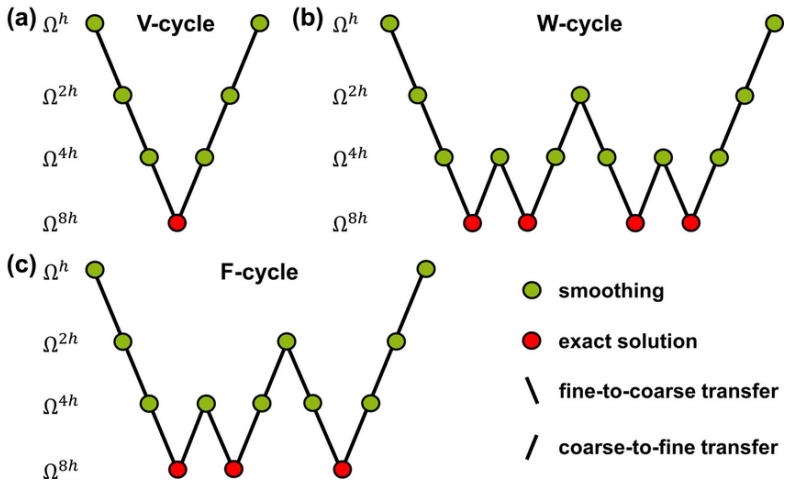

Figure 1: Multigrid cycles

- **Restriction:** computes the residual of the current problem and projects it onto the coarser grid. Examples of such projections are injections and full weight averages of neighbouring nodes;
- **Prolongation:** prolongs the error onto the finer grid interpolating the missing nodes;
- **Direct solve:** solves exactly the coarsest problem with a direct method.

The building blocks are aggregate in recipes called "multigrid cycles" that differ in cost of computation and accuracy of correction (see figure 1)

2.1 Efficacy of multigrid

The error correction is so useful because of a property of smoothers and grid transfers. Applying only smoothing of the fine grid will eventually converge but extremely slowly. This is because smoother application is equivalent to multiplying the current error by an iteration matrix. This iteration matrix has a spectral radius less than 1 and that's the reason the error converges to zero

$$||e^{(n)}|| = ||B^n e^{(0)}|| <= ||B||^n e^{(0)}$$

but some eigenvalues are very close to 1 and as n grows they get even closer. If the initial error has some components along the eigenvectors with eigenvalues close to 1 (something that one should expect since the initial error is unknown) the first iterations will attenuate the other error components and the iteration will stagnate. There is a mathematical phenomena for which these stubborn components when

projected onto a coarse grid will become easier to attenuate by the corresponding grid smoothers.

2.2 Challenges of multigrid

Geometric multigrid relies on the structure of the grids being one the refinement of the other. This makes the implementation and the error analysis easier and it's by far the most common choice but constrains the number of grid points to:

$$n = 2^k m + 1$$

Also the choice of the grid transfers is critical, they need to couple with the smoothers to actually attenuate the stubborn components of the error and not just shuffle them around. The grid transfer operator are responsible of how the error will be relaxed on coarser grids because of the Galerkin relation:

$$A^{2h} := I_h^{2h} A^h I_{2h}^h$$

3 Program structure

The project proposed is a C++17 application, although the core features are compatible with C++11, with an `Eigen` dependency to solve linear problems with direct methods. Given a PDE problem on a square computational mesh and the corresponding discretized operator it computes the solution iteratively via the multigrid method with complete customization over cycle, smoother and grid transfer operators. The goal of the project was to produce an implementation that could be extended to solve not only the Poisson 2D problem, but any 1D and 2D problem that admits a finite difference discretization. This is empowered by a polymorphic design that will be further explained in the following sections.

3.1 Essential abstract classes

This section will be of interest to anyone navigating the codebase or planning to expand on it. All instances of "abstract class" in this document are referring to the object oriented paradigm definition. In other words one might think it as a blueprint of functionality that needs to be implemented by a concrete class.

3.1.1 `class Problem`

This class is responsible of containing all the data about a particular PDE problem like the number of nodes, the geometry of the computational mesh and so on.

The number of nodes is essential to handle memory allocations by other parts of the codebase, so a getter method is required. The problem has control over the discretization method used via the `get_discrete_operator(int level)` method that returns a `DiscreteOperator` object. The problem could be discretized on multiple grid levels as a requirement of the multigrid method, so it's natural that the `Problem` class has the appropriate logic. While operator restriction can be a challenging task, the current specializations of `Problem` rely on the fact that A^{2h} is the operator of the problem discretized on a $2h$ grid. The property is investigated in section 4.1 and it's a direct consequence of dealing with square grids. The class also holds ownership of the right hand side vector of the linear problem. This decision was made to save memory in the case of solver comparison. When two concurrent solvers are working on the same problem (so also same size) it would be a waste to compute two rhs vectors because they are the same. A problem can also initialize the solution when Dirichlet boundary conditions are involved with the method `set_initial_approximation()`

3.1.2 `class DiscreteOperator`

The `class DiscreteOperator` knows how to:

- compute the residual given the right hand side and the solution
- relax the solution applying one of the smoothers
- produce an `Eigen` compatible sparse representation

This class really captures the variability of the PDE landscape. It has (implicit) information about the mesh structure and can discriminate between inner and boundary nodes and can successfully apply the smoothers on the solution. It's been designed with little constraints as possible to admit many possible implementations for example:

- stencil based implementations, implemented in the program
- sparse matrix based implementation

It requires a sparse matrix representation because the `Eigen` direct solvers need such structure to operate. The stencil based approach was designed for pedagogical purposes but can at most have a performance advantage over sparse matrices for structured square meshes. Real problems favour the sparse approach but then the project would have become an Eigen wrapper.

3.1.3 `class IterativeSolver`

A simple class that has ownership of the solution, has logic for solving a problem iteratively meaning:

- initialization of the solution
- apply an iteration step
- compute the residual and assess termination conditions

It is constructed with a pointer to a `Problem` class, which means that it's agnostic to the details of the single problem and the single discrete operator. The `MgSolver` class is derived from `IterativeSolver` and apart from the constructor, it had to override only the `step()` method to be considered complete.

3.2 Program flow

To wrap up the description of the classes here is a common flow in the program:

1. declare a `Problem` class
2. declare the `MgSolver` class with the problem pointer
3. call the `MgSolver.solve()` method

Other details can be interpolated looking at the examples program in the `/examples` folder.

3.3 Usage details of `MgSolver`

The multigrid solver is constructed with a pointer to a `Problem` class, a recipe of the cycle and the grid transfer operators. Again the polymorphic design makes the implementation suitable for every problem and discrete operator combination. The construction of the solver might throw some exceptions because the grid geometry doesn't allow the number of requested subdivisions, this halts the program and prints an error message

4 Design decisions

Since the multigrid method was uncharted territory for me, in light of the challenges of the hands-on project, I decided to structure the project with incremental steps and continuous feedback about program correctness

4.1 Operator restriction

The bug in the hands-on project was in the operator restriction. A thoughtful analysis was made to eliminate any doubt around the topic, it involved:

- study of A Multigrid Tutorial[1]
- jupyter notebook analysis at `analysis/restriction.ipynb`

The notebook employs a computer algebra system to produce the Poisson operator and its restriction using the Galerkin condition introduced above

$$A^{2h} := I_h^{2h} A^h I_{2h}^h$$

For the 1D Poisson problem it assessed that with full weight restriction and linear prolongation the restricted operator is the discretized operator on a $2h$ grid. In the 2D case every restriction and operator choice I tried didn't yield the expected operator, but experimental results show convergence despite the theoretical unfoundedness.

4.2 Convergence tests

As mentioned before, the second order discretization of the Poisson equation yields numerical solution that converges quadratically in h to the exact solution. The convergence tests are excellent for assessing the correctness of implementation:

1. come up with the exact solution
2. produce the forcing term $f := -\Delta u$
3. convert mathematical functions to code

This process is error prone and tedious, so the average programmer will implement only a couple of test cases and will forget to check the convergence after every major refactor or feature addition. The project implements an automatic path to (2) and (3) via cmake and python: the user writes the exact solution as a symbolic expression in `analysis/poisson_exact_solution.py` and the cmake build invokes the commands to produces error-free c implementations of the u and f. Another cmake path builds the example convergence path and plots the results like figure 2.

PS: I'm an average programmer too, so this integration happened too late when it was needed

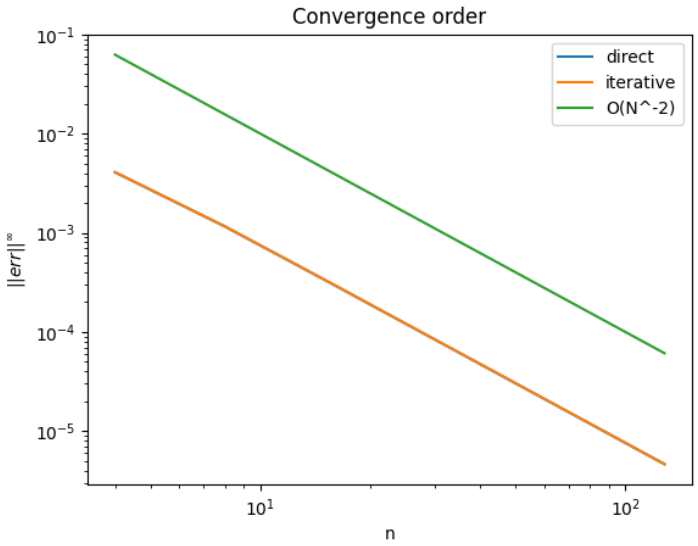

Figure 2: Discretization error for increasingly bigger 1D Poisson problems

4.3 General automation

Another aspect that improves developer experience is to never leave the keyboard. The workflow of running examples, collecting experimental data and plotting it is entirely automated via custom cmake rules and python scripts. The experimental results are always in a ascii encoded csv format and that makes the plotting scripts portable across different projects. Section 6 presents the cmake custom rules for invoking examples

4.4 Stencil approach

Stencils were chosen as a possible performance improvement over sparse matrices because the data access pattern (for example how many memory accesses per row to compute a single element) is known at compile time. Besides performance reasons I wanted to stretch the `ThreePointStencil` class to solve the forced RC circuit problem:

$$\begin{cases} u'(t) = -\tau u(t) + f(t) \\ u(0) = u(T) \end{cases}$$

The peculiarity of the problem is the periodic boundary conditions, which require a computational mesh that is topologically connected at the extremes (a one dimensional torus, S^1). The implementation of `class ThreePointPeriodicStencil` is

the proof that the real problem logic lies in the operator class and the problem class is just a convenient wrapper.

5 Multigrid implementation

With the class hierarchy described above, the multigrid solver was only a matter of filling the gaps:

- own the solution, rhs, operator and residual of the coarse problems
- implement grid transfer operators
- glue all together in the `step()` function

There was a minor inconvenience about the rhs locality because at the finest grid the rhs is owned by the problem but the coarse rhs are internally owned and must be writable when restricting the residual. The solution was to build a memory map (a vector of pointers) that for each level would give the solution, rhs and residual memory. To account for const/non-const shenanigans the class `PointerVariant<T>` has method for treating both const and non-const pointers with pointer arithmetic and conditional casting to mutable pointer.

5.1 Smoother landscape

The relaxing procedure is delegated to the `DiscreteOperator` and is essentially variation on the update order of a fixed iteration formula, the update order is what makes the smoother parallelizable or not:

- Jacobi, simultaneous update, maximum parallelization;
- Gauss Seidel, sequential update, not parallelizable;
- Red Black Gauss Seidel, partitions the grid into red and black nodes update in parallel the red entries then the black ones

Jacobi smoother has the problem of needing temporary memory to store the result (and so needing a full copy at every smoothing step in simple implementations) but it also has the worst smoothing performance above the smoothers, so no attention was put to it.

5.2 Parallelization options

There is room for cpu OpenMP parallelization due to the parallel nature of the computation, every basic block can be separately parallelized. This could mean a possible linear speedup in the number of threads but the direct solve cannot accelerated, this implies a sublinear speedup as the benchmarking section of this document will present.

5.3 Kernel fusion improvement

Multigrid building blocks are implemented in separate functions, each opens a OpenMP parallel region and there is a little overhead. The project implements a variation on the multigrid cycle where:

- the computing of the residual and its restriction is done in the same function with limited memory transactions
- the prolongation of the error and its correction is done in the same function with no temporary memory transactions

The algorithm if the optimized versions are a little messy and obviously break some encapsulation because they fuse scopes, but the benchmarks will assess a 10-15% improvement over the non fused version.

6 Examples

Illustrates common uses of the library and shows common multigrid phenomena

6.1 Convergence 1D

Solving the isotropic Poisson problem was the first feature implemented in the project. It shows an interesting numerical phenomena due to floating point round off. When relaxing the solution the stencil operation would be

$$u_i = \frac{h^2}{2}(f_i + \frac{u_{i-1}}{h^2} + \frac{u_{i+1}}{h^2})$$

But then encounters floating point errors due to dividing with a very small denominator and possibly some absorption errors. A more sensible formula would be

$$u_i = \frac{h^2 f_i + u_{i-1} + u_{i+1}}{2}$$

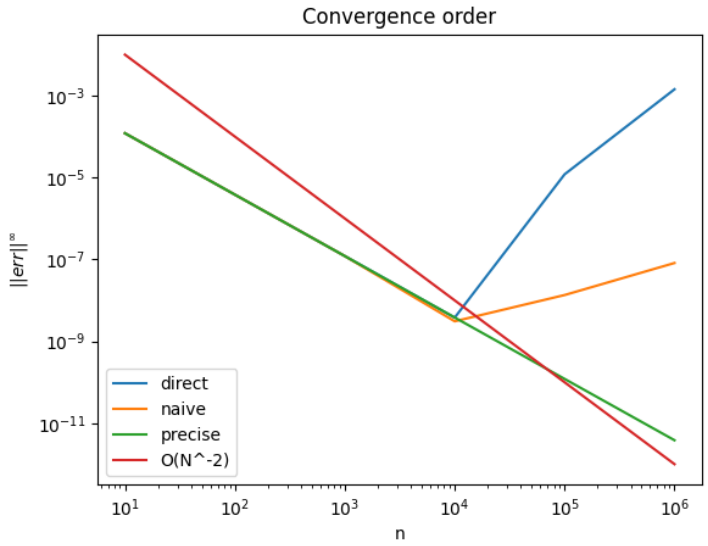

Figure 3: Comparison of the discretization error of a 1D Poisson problem given a "naive" formula and a better "precise" formula

The same reasoning can be applied to residual calculation. It happens that for one dimensional problems defined in the interval $[0, 1]$ the choice of h is critical because if h is too small then $h^2 < 10^{-16}$ and this should light danger in the numerical analyst. A demonstration of the phenomena is at `examples/example_convergence_1d.cpp` where two Poisson problems are solved with two formulas equivalent in exact arithmetic but one clearly better than the other. Figure 3 shows that there is loss of convergence when the step is too small, the same thing would not happen if the domain is much larger (same number of nodes is required). Also a solution obtained by a direct method with the naive formula is included. This example can be shown by `make convergence_1d`

6.2 Convergence 2D

Figure 4 is a simple convergence test, using the symbolically generated functions. Compares the multigrid solution error with the direct solution error. A remark: due to the fill-in phenomenon the direct solver `Eigen::SparseLU` stalls when trying to factorize a problem bigger than 127×127. In possible future work one could experiment different solvers from other libraries or a completely different approach like FFT based solvers. There is the expected quadratic convergence so all good, can be called with the command `make convergence_2d`

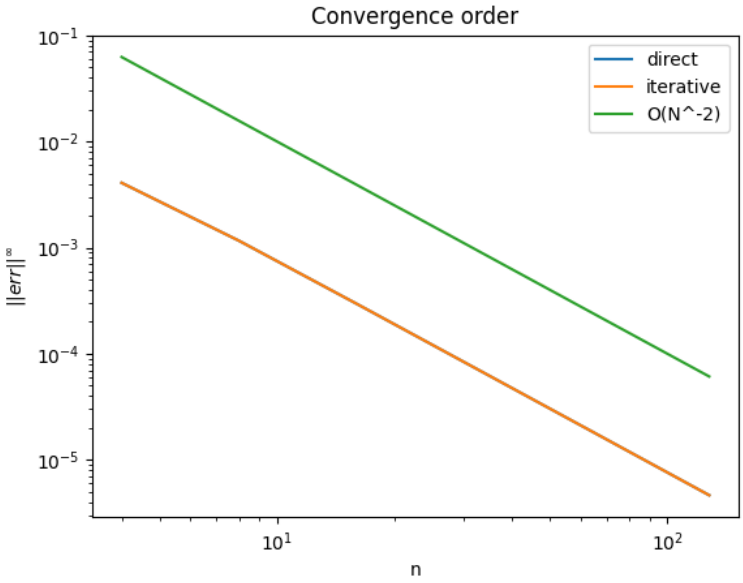

Figure 4: Convergence test for a 2D isotropic Poisson problem

6.3 Circuit

Solves the periodically forced RC oscillator with both direct solvers and iterative ones. Plots the input signal and the output signal, for the electronics enthusiasts out there it shows an attenuation of the signal and a phase shift characteristic of an RC filter while still maintaining periodicity of the output. The example program prints also the solve time, when the problem is big ($N > 10^6$) the multigrid is the fastest to solve. Can be called with the command `make circuit`

6.4 Multilevel

Compares the the convergence speed in terms of residual norms for different depth V-cycle multigrids, theory expects the 2 level to be the fastest to converge and then the 3 level and so on. Experimental results show that they are equivalent and at most the difference to convergence is of 1 iteration. Figure 6 doesn't show 2 level and 3 level cycle results because the problem is too big for the direct solver to handle. One important remark: multigrid is a scalable method, the iteration to convergence are minimally dependent on problem dimensionality and definitely not dependent on mesh granularity so expect to see the same behaviour for different problem size. `make multilevel`

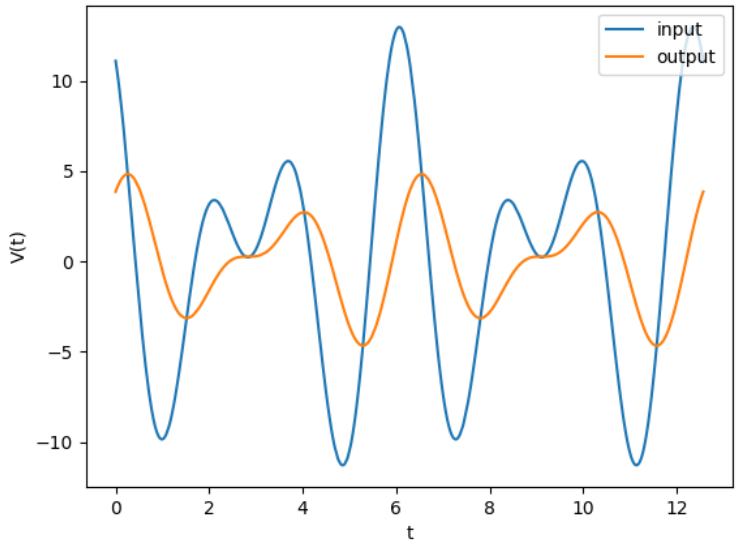

Figure 5: Filter response of an RC circuit when submitted a periodic input signal

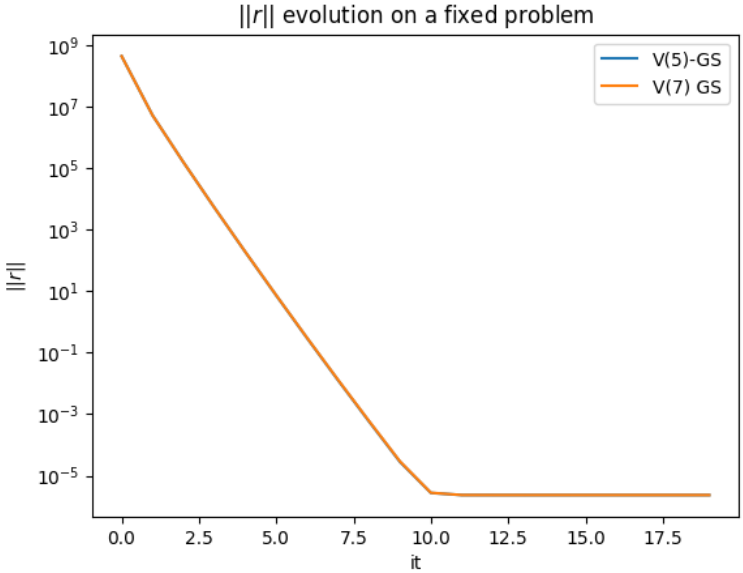

Figure 6: Multilevel convergence performance on a fixed Poisson 2D problem

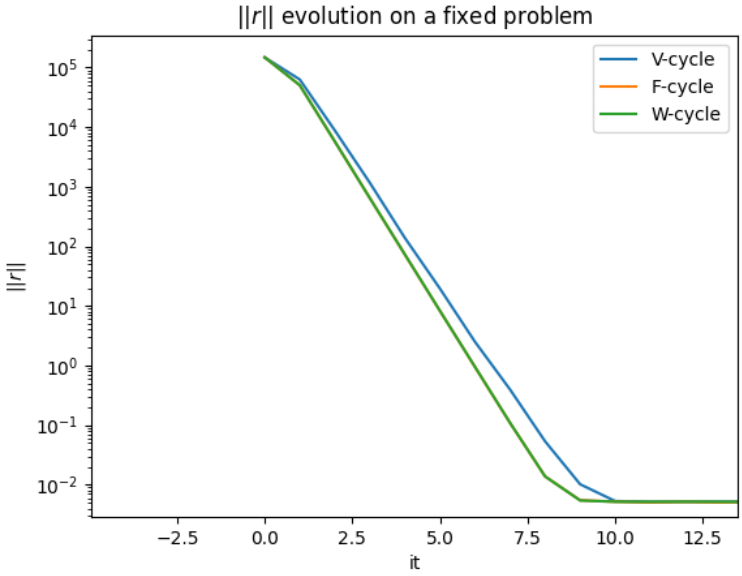

Figure 7: Comparison of V, W, and F cycle (in this plot W and F are overlapping)

6.5 Cycle

Figure 7 compares the convergence speed of different multigrid cycle with the same depth on a fixed problem. They only differ for a couple iterations to convergence and the V-cycle is the most performant because it does the least amount of work. `make cycle` to experiment with this example.

6.6 Smoother

The most interesting (and the most confusing) plot among all. Figure 8 shows the residual behaviour of same depth V-cycle multigrid solvers, where the only difference lies in the smoother choice and the restriction operator (default operator is the full weight). It shows that Jacobi is not a suitable smoother for the problem, SOR improves it a little bit and unexpectedly the injective restriction works well even when not recommended by the theory. It is the reason I still have some doubts about the correctness of my implementation but I did my best. `make smoother`

7 Benchmarks

Benchmarks are difficult to get straight, there are frequency scaling problems, empty cache effects and might even test the system in a unreal situation. I de-

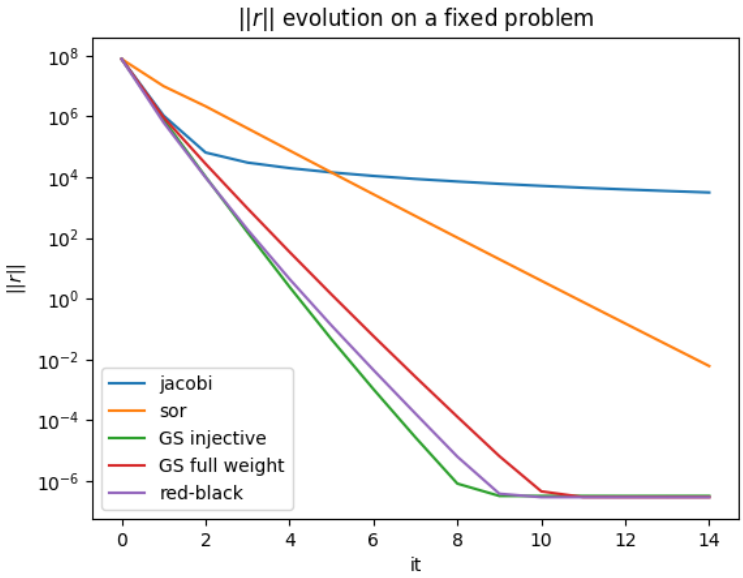

Figure 8: All round demonstration of grid operators and smoothers collaborating to cut the error

cided to use the google microbenchmarking library as my daily driver because it provides logic for the automatic repetition of runs and provides a good result format from which with some Python magic I can parse results out and produce a fancy matplotlib plot. Now given the extremely close performance (in iterations until convergence) of Gauss Seidel and red black based multigrid solvers I decided to profile the step time

7.1 Scalability, `make benchmark_vcycle`

The Poisson 2D problem scales linearly with the number of nodes so quadratically with the number of nodes in a row for square meshes.

Figure 10 shows a clearer picture of the performance gain of non parallelized red black over gauss seidel

7.2 V, F, W cycles, `make benchmark_cycle`

Figure 11 is proof that the V cycle is the best

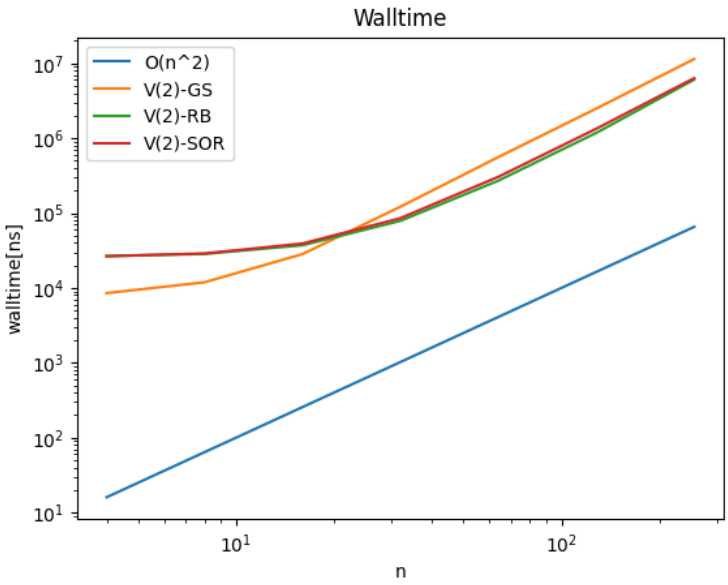

Figure 9: Scalability test of multigrid variations

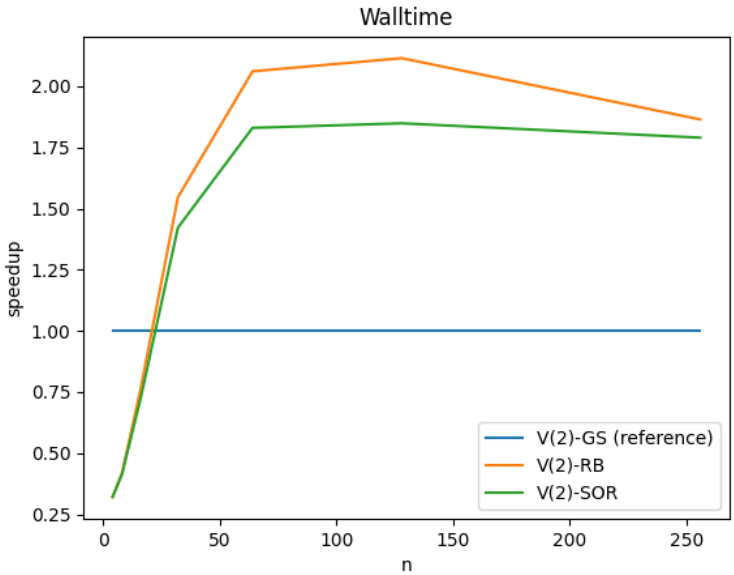

Figure 10: Speedup for redblack over the scalar Gauss Seidel

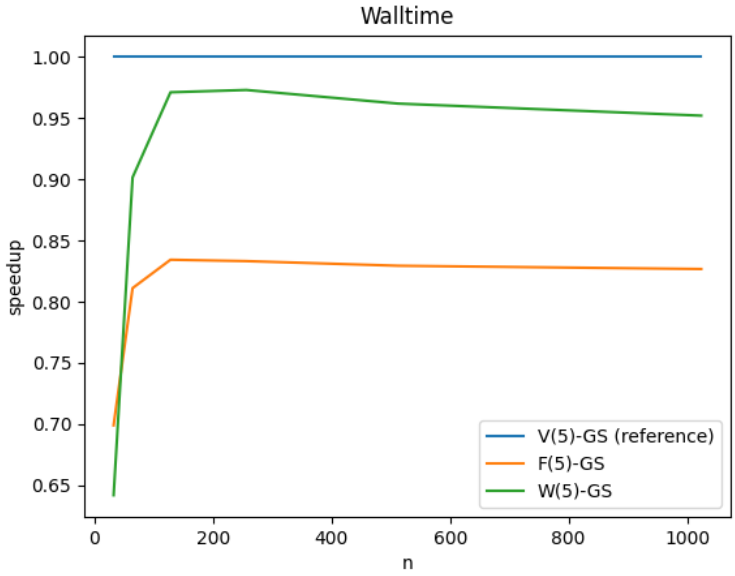

Figure 11: Speeddown for W and F cycles with respect to a same depth V cycle

7.3 OpenMP parallelization, `make benchmark_omp`

Figure 12 shows the speedup of multithreaded red black multigrid solvers over the single threaded one. Expected sublinear speedup to to the presence of overheads and the serial direct solver. On my 4 core machine is compatible that the 8 threads are comparable with the 4 threads.

7.4 Kernel fusion, `make benchmark_fused`

Figure 13 shows the relatively unstable speedup improvements of the kernel fusion technique. Keep in mind that this technique disrupts the architecture of the program and makes many assumptions on the problem at hand so it's not very portable. Nevertheless there are improvements without throwing more hardware at the problem, which is nice.

8 Conclusions

This was the result of a one man effort over an entire month of work, I definitely won't repeat the mistake of working alone on such big projects, it was tough. I hope that part of this project (especially the theoretical part of the presentation) will be useful to students studying numerical linear algebra and implementing

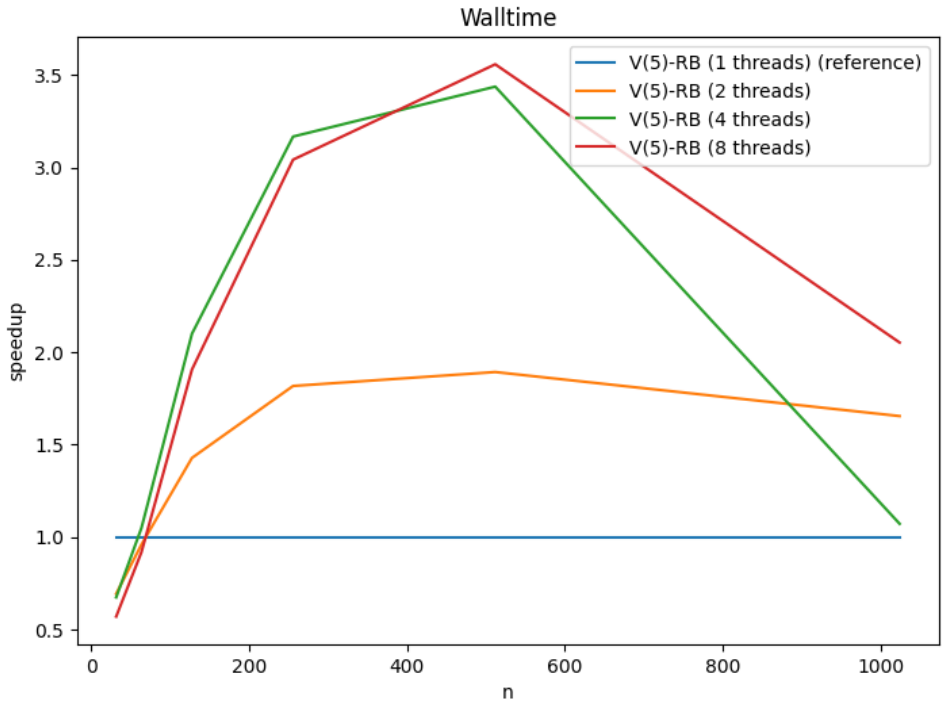

Figure 12: Speedup of red black multigrid solver step function when called with different number of OpenMP threads

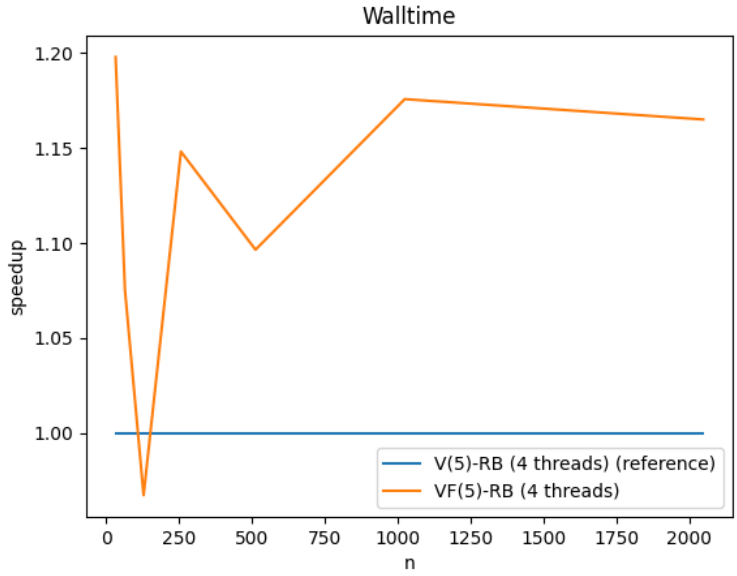

Figure 13: Speedup of red black multigrid solver with kernel fusion optimizations

multigrid method. Future work will involve a better understanding of a method so powerful from a computational point of view because coarse grids are a way to accelerate information exchanges between far nodes and this is truly an exascale idea.

9 Bibliography

1. William L. Briggs, Van Emden Henson, and Steve F. McCormick, A Multigrid Tutorial, Second Edition, Siam, (2000).