CME 335 Project Final Report: A nested dissection implementation with updating

Victor Minden*

March 18, 2015

1 Introduction

Many partial differential equations (PDEs) arising from mathematical models of physical phenomena, when discretized using the finite-difference or finite-element method (for example) with fields represented on a Cartesian grid, lead to linear systems with highly structured sparse system matrices. For example, consider a general elliptic PDE (and some given boundary conditions) of the form

$$-\nabla \cdot (a(x)\nabla u(x)) + b(x)u(x) = f(x), \quad x \in \Omega \subset \mathbb{R}^d \tag{1}$$

where $d = 2, 3$, u is the unknown field of interest, f is the source term, and a and b are material parameters of some sort. Concretely, with $a(x) \equiv 1$ and $b(x) \equiv 0$, this reduces to Poisson's equation, of great importance in mathematical physics.

We consider a particular discretization of (1), though not that really a broad class of discretizations fall into this form. With $d = 2$, let $x_{i,j} = (ih_{x_1}, jh_{x_2}) \in \mathbb{R}^2$. Then we can consider using a 5-point finite difference discretization of the Laplacian operator, leading to the discrete equations

$$\left(\frac{a_{i-1/2,j} + a_{i+1/2,j}}{h_{x_1}^2} + \frac{a_{i,j-1/2} + a_{i,j+1/2}}{h_{x_2}^2}\right) u_{ij} + b_{ij}u_{ij}$$
$$-\frac{a_{i-1/2,j}u_{i-1,j} + a_{i+1/2,j}u_{i+1,j}}{h_{x_1}^2} - \frac{a_{i,j-1/2}u_{i,j-1} + a_{i,j+1/2}u_{i,j+1}}{h_{x_2}^2} = f_j$$

for each unknown (correction terms near the boundaries depending on boundary conditions)[1]. More generally, we can consider a stencil parameterized by a "width" parameter w to obtain

$$f_j = \left(\frac{1}{h_{x_1}^2}\sum_{p=0}^{w-1}(a_{i+p+1/2,j} + a_{i-p-1/2,j}) + \frac{1}{h_{x_2}^2}\sum_{p=0}^{w-1}(a_{i,j+p+1/2} + a_{i,j-p-1/2})\right) u_{ij} + b_{ij}u_{ij} \tag{2}$$
$$-\frac{1}{h_{x_1}^2}\sum_{p=0}^{w-1}(a_{i+p+1/2,j}u_{i+p+1,j} + a_{i-p-1/2,j}u_{i-p-1,j}) - \frac{1}{h_{x_2}^2}\sum_{p=0}^{w-1}(a_{i,j+p+1/2}u_{i,j+p+1} + a_{i,j-p-1/2}u_{i,j-p-1}).$$

In other words, these are stencils corresponding to higher order discretizations of the Laplacian (note that $w = 1$ gives us the 5-point stencil approximation in 2D, $w = 2$ gives the 9-point stencil, *etc.*). In practice, one would not take w very high, but frequently applications engineers will choose $w = 1$ or $w = 2$. We can rewrite this in the more abstract form $Au = f$, where if a is positive and bounded away from zero then we

*Institute for Computational and Mathematical Engineering, Stanford University, Stanford, CA 94305. Email: vminden@stanford.edu

[1]Apologies for the awful line break

see that A is positive-semidefinite, sparse, and highly structured (in fact, pentadiagonal). In 3D, (2) extends trivially as one would expect and we obtain a structured matrix that is now heptadiagonal.

While we focus here on the elliptic case, we note that there are two more similar problems that we are interested in handling. The first is the case of parabolic differential equations, such as the heat equation

$$u_t(x,t) - \nabla \cdot (a(x)\nabla u(x,t)) = f(x,t), \quad (x,t) \in \Omega \times \mathbb{R}^+. \tag{3}$$

Here we see that after backward-discretization in time we need to solve a system looking very much like our Laplacian system but perhaps even more positive definite (a multiple of the identity has been added to the diagonal).

A second similar problem is the case of the convection-diffusion equation,

$$u_t(x,t) - \nabla \cdot (a(x)\nabla u(x,t)) + \nabla \cdot (\mathbf{v}(x,t)u(x,t)) = f(x,t), \quad (x,t) \in \Omega \times \mathbb{R}^+, \tag{4}$$

of which the heat equation is a special case. What is intriguing about (4) is first that it generally leads to an indefinite system matrix, A, and second that it frequently leads to an asymmetric system matrix due to upwinding of the advection term. In general, the multifrontal nested dissection we describe in this paper is not guaranteed to work on this PDE, but we would like to try it out in practice.

1.1 Solution via nested dissection

To solve the regular positive semidefinite structured systems that come out of 2D or 3D PDEs, one important direct method is the method of nested dissection due to George [1]. In the context of what we discussed in class, nested dissection is essentially a fill-reducing ordering for sparse Cholesky factorization of a system corresponding to the discretization of a PDE on a grid that uses physical information about the grid to intelligently choose separators. The factorization can then be formed efficiently using dense frontal matrices just as in the more general multifrontal method we discussed in class.

For brevity, we assume that the reader is familiar with the multifrontal method as discussed in the context of class, and otherwise direct any interested readers to the handy description by Liu [3]. We will henceforth be loose in the distinction between the general multifrontal method and the multifrontal method with nested dissection order on grid graphs.

For our purposes, we will consider the (hyper-) cross separators that partition a rectangular prism domain in $\mathbb{R}^d$ into 2^d subdomains at each step, which should be optimal up to a constant factor. For the elliptic and parabolic problems where the system matrix in question is positive definite, the eliminations corresponding to this frontal ordering will all involve invertible pivot blocks and thus this leads to a numerically stable and well-posed procedure. This is not necessarily the case for the mixed hyperbolic advection-diffusion equation.

1.2 Modified factorizations

In an equation such as (1), it may be the case that one forms a discretization such as (2), constructs a factorization of the operator and solves the necessary right-hand-sides, and then modifies some of the material parameters a or b, thus changing the system matrix. In [4], Minden *et al*consider similar updates for tree-based factorizations of integral equations and show that in the specific case of *local updates* where the modified material parameters are spatially-localized, the factorization can be updated without refactoring every node of the elimination tree. Rather, because of the inherent spatial partitioning in the multifrontal method, it is possible to get away with only refactoring those leaf-level fronts that have explicit modifications due to changed material parameters, and then trace their impact up the elimination tree and refactor only certain paths to the root.

Under certain assumptions, the above factorization updating process is asymptotically more efficient in the case of the hierarchical interpolative factorization (HIF) of Ho & Ying [2], but it is worth noting that this process always requires refactoring the root node of the elimination tree. In the case of the multifrontal method, refactoring the root node is asymptotically just as expensive as refactoring the whole tree, and therefore one should not expect more than a constant factor (if that) improvement. However, this same updating process *is* possible.

1.3 Contribution

The contribution of this project is an implementation in C++ of the nested dissection method with cross separators for problems in 2D or 3D, with careful modification to allow for updating. These choices will be described later in the body of this document. In the interest of having this be usable code for active researchers, we have the following capabilities in mind:

- Allow for an arbitrary number of points in each coordinate direction (*i.e.*, N_x, N_y, N_z not constrained to be equal or 'power-of-2-plus-1' or the like).
- Handle the non-zero structure arising from Laplacian stencils based on the central-difference formulas in each coordinate direction of arbitrary order, *i.e.*, with width parameter w as a runtime parameter.
- Allow for the possibility of unsymmetric discretizations with symmetric structure, arising, *e.g.*, from upwinding.
- Admit updating due to local modification of material parameters.

The first item above is the largest headache in moving from theory to practice, as recursively subdividing a domain is a lot simpler when the domain is self-similar with respect to the recursion and has a degree of homogeneity. However, in practice domains are not dyadic. The second item above is important largely because it increases separator width but is otherwise rather simple to implement, and many research groups insist on higher-order Laplacian discretizations. The third item above is important for exploring advection-diffusion, as described previously. The final item concerns the updating strategy previously sketched.

This code is meant to serve as the basis for a code for updating HIF, similar to that of [4] but applied to sparse discretizations of PDEs rather than boundary integral equations. The code is available on BitBucket in a git repository already shared with Prof. Poulson, and will be made available to the public upon completion of the HIF portion, which is future research.

2 Summary of work

The implementation of this project has required a few different components. First, because the target community is computational physicists, the language of choice was to be something low-level, and we ultimately settled on C++. To keep the package lightweight, the necessary functionality is implemented as a template library to allow ultimately for both real and complex matrices. For the purposes of this course, we restrict our discussion to the real case, but note that the functionality for complex matrices is theoretically present and simply as of yet untested. The library requires only BLAS and LAPACK as dependencies to stay lightweight, and uses the CMake build system.

2.1 Dense linear algebra

Because the language for this project was C++ and not a higher-level language such as Julia or MATLAB, dense linear algebra is not a core language feature. As such, we began by constructing C++ template classes for dense matrices and vectors using lightweight wrappers to call the necessary BLAS / LAPACK routines.

The core dense linear algebra functionality necessary essentially comes down to the following set of routines:

- GEMM (for mat-mat multiplication with optional (Hermitian) transpose flags)
- AXPY (for vector addition operations)
- NRM2 (to measure error)
- SCAL (not actually necessary in this routine, but useful to have)

- GEMV (for mat-vecs with optional (Hermitian) transpose flags)
- GETRF (to form a PLU factorization to avoid explicit inverses)
- GETRS (to solve with above PLU)

As mentioned, we are hoping in general to be able to try this on systems with, for example, upwinding, and so in this preliminary implementation we have only wrapped the general matrix routines and not routines that take advantage of Hermitian structure. However, because we have constructed a unified interface to the linear algebra routines, it will not be difficult to add the additional routines and specialize for performance boosts later on.

Beyond the dense linear algebra functionality, we also constructed a sparse matrix class for representing sparse CSR-format matrices and applying them, viewing them, *etc.* This was useful for debugging and for checking errors in our factorization early on.

2.2 Preprocessing

Naïvely, a multifrontal method on a structured grid simply consists of the following process:

1. Recursively subdivide the domain until the leaf boxes are sufficiently small.
2. Eliminate the interior of each box and apply the Schur complement update to the surrounding separators.
3. If there are more levels, step up a level and go to previous step.

Of course from an implementation standpoint, the process above unfortunately glosses over some very important details. First, subdividing the domain is not an entirely trivial task when the domain is rectangular (has a different number of unknowns along each coordinate direction) and does not have a nice power-of-two-related structure, since then it is not self-similar. Second, when forming Schur complement updates to the separators, it is undesireable to actually modify the entries in a large sparse matrix since changing the structure of a sparse matrix can be very inefficient and we would waste a lot of time indexing.

Now, we could of course use a tool like METIS to perform the subdivision of the domain, but in the interest of writing code that is usable later for our HIF implementation we have not done this. Thus, we instead begin with a large amount unfortunate preprocessing to figure out the degrees of freedom (DOFs) that belong to each leaf level box, storing them in a data structure with signature

```
struct Box {
            IdxVec intDOFs;                   // eliminate all interior,
            IdxVec bdDOFs;                    // keep boundary DOFs
            LinAlg::Dense<Scalar> A22;        // matrix restricted to interactions
            LinAlg::Dense<Scalar> A22Inv;     // not explicit inverse of A_22, but applies PLU
            LinAlg::Dense<Scalar> XL;         // A_12 * A_22_inv
            LinAlg::Dense<Scalar> XR;         // A_22_inv * A_21
            LinAlg::Dense<Scalar> SchurComp;  // -A_12 * A_22_inv*A_21
            LinAlg::Dense<Scalar> A11post;
}; // struct Box
```

Here, the struct `Box` knows the interior DOFs to a box (those that we will eliminate), the boundary DOFs to a box (those separator DOFs at this level that will recieve Schur complement updates), and then certain blocks necessary for applying the elimination matrices. It is also worth mentioning the `A11post` variable: this is the self-interactions of the separator DOFs after the Schur complement update has been applied. While storing this information at each level is not necessary in the general case, this is critical for allowing our factorization-updating method to work: in order to only follow a single path of updates up the elimination tree without looking at the factorizations we don't need to redo, it is necessary to be able to reconstruct the "state" of the interactions between DOFs at any given level of the elimination tree.

2.3 Factoring

The core flow of the factoring routine, as discussed, is simply to loop over the elimination tree from bottom to top and, for each box in the level, to determine the proper information to populate the `Box` data structure and then do a simple step of elimination. The dense linear algebra can be very simply expressed in code: at level `ell` we simply perform the operations below.

```
Dense<Scalar>& A11     = boxData_[ell][boxIdx].A11post;
Dense<Scalar>& A22     = boxData_[ell][boxIdx].A22;
Dense<Scalar>& A22Inv  = boxData_[ell][boxIdx].A22Inv;
Dense<Scalar>& XL      = boxData_[ell][boxIdx].XL;
Dense<Scalar>& XR      = boxData_[ell][boxIdx].XR;
Dense<Scalar>& S       = boxData_[ell][boxIdx].SchurComp;

// Already pulled up A22 in setup, so just invert it
A22Inv = A22;

/* Very possible that XL can't be both LHS and argument */
Dense<Scalar> A12 = XL, A21 = XR;

LU(A22Inv);
XL = A12;
LUSolveT(A22Inv, XL);
XR = A21;
LUSolve(A22Inv, XR);

S    = A12 * XR;
S    *= -1.0;

// Update the cliqueified interaction between separator DOFs
A11 += S;
```

The linear algebra itself is not complicated here – the programming challenge is properly populating the `Box` data structure. Luckily, there are some simple rules to determine the correct state of the interactions at each level which we only sketch here in the interest of brevity.

Consider the 2D case. Then, boxes are subdivided according to Figure 1. Assuming the blue boxes comprise the leaf level, all blue boxes will have all DOFs eliminated and form Schur complement updates to the purple separating edges. Note that based on our assumption on the form of the Laplacian discretization, the green points will not recieve Schur complement updates. This implies that at the parent level, all interior points of the parent with *modified* interactions come from the "separator children". The green points will also be interior to the parent, but their interactions have not been modified and thus can be extracted directly from the original sparse matrix.

2.4 Updating

Given that each `Box` data structure already contains the `A11post` matrix containing the states of the interactions at each level (at worst a factor of 2 more expensive in memory usage), for the updating procedure only a very small modification is necessary. We implemented a function `Update` with prototype

```
void Update(const LinAlg::Sparse<Scalar>& A, std::set<int> modifiedDOFs);
```

This function allows modifying of the discretization matrix A and requires knowledge of the DOFs corresponding to rows or columns with modified entries. We then look up which leaf-level boxes own these DOFs and begin refactoring paths up the elimination tree. Thus, at each level, we only loop over a smaller set of "marked" boxes and re-eliminate those corresponding DOFs as opposed to looping over all boxes.

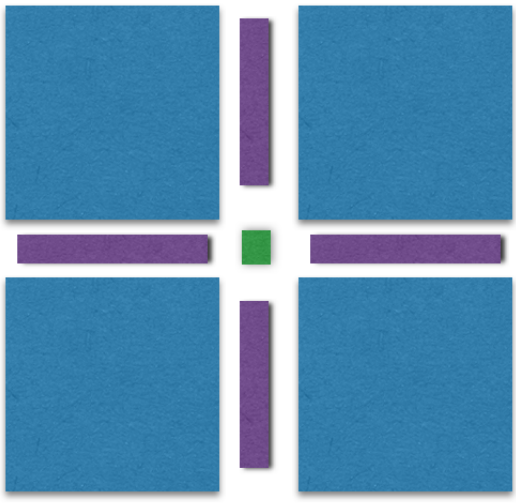

Figure 1: An example of how the domain is subdivided in 2D. At each step, a parent box is broken into 4 child boxes (blue), 4 edges comprising a separator (purple) and a set of points that do not below to either child boxes or child separators (green). Not shown: the separators separating these blue boxes from the children of neighbors of the parent box.

3 Numerical Results

We have a number of different tests that we've run to analyze the performance of our solver, with an eye towards timing, accuracy, and robustness.

All computations were performed on a single core (without parallelization) of an Intel Xeon E7-4820 CPU at 2.0 GHz on a 64-bit Linux server with 256 GB of RAM.

3.1 Example 1: Scaling results for homogeneous media

To begin, we consider (1) with $a(x) \equiv 1$ and $b(x) \equiv 0$ and uniform Dirichlet boundary condition $u_{\partial\Omega} \equiv 0$. This is obviously an idealized physical model, which we will use to investigate timing. Note that this is not sweeping anything under the rug since the timing of the multifrontal method should not depend on the "hardness" of the problem, only the solution accuracy will.

Using separators of width 1 in 2D, we obtain a pentadiagonal finite difference matrix to which we apply our solver for varying N. To investigate scaling in 2D, we set $N_x = N_y = \sqrt{N}$ and look at the factor time and apply/solve time as we increase N. All data points for plots and tables are obtained by averaging over 3 different trials so as to somewhat mitigate fluctuation in runtime due to the operating system or other processes, but this did not prove to be necessary as the runtimes were almost identical.

We see in Figure 2 the runtime results for this example as we vary N between roughly 1K and 16M unknowns. The factor time begins scaling better than expected but around the $N = 1M$ mark starts to exhibit the expected $\mathcal{O}(N^{3/2})$. On the other hand, the applies and solve follow exactly the expected scaling of $\mathcal{O}(N \log N)$.

To show that we're not sweeping anything under the rug here, we also show in Figure 3 the relative apply error and solve error for this example. As would be expected, the apply error stays under $1e-15$ and grows only very slowly with N. However, the solve error does creep up dramatically as the number of unknowns is increased, with an observed scaling of $\mathcal{O}(N)$. Theoretically, we know that the conditioning of this finite-difference Laplacian goes something like $\mathcal{O}(N)$, which we verify in Figure 4, so this makes sense – we are observing limits of floating point arithmetic due to numerical conditioning.

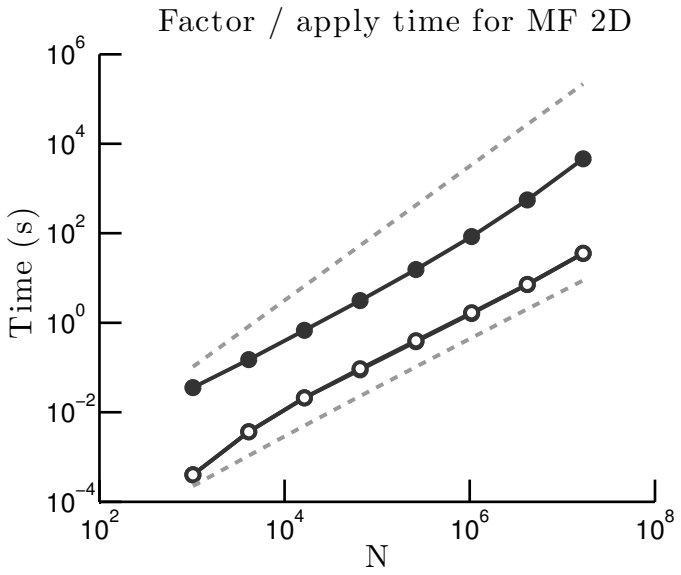

Figure 2: Timing results for Example 1 in 2D. Black circles correspond to factor time, white circles correspond to apply time, and grey circles correspond to solve time. The top guideline is $\mathcal{O}(N^{3/2})$ and the bottom guideline is $\mathcal{O}(N \log N)$. Note that the apply and solve curves lay on top of each other.

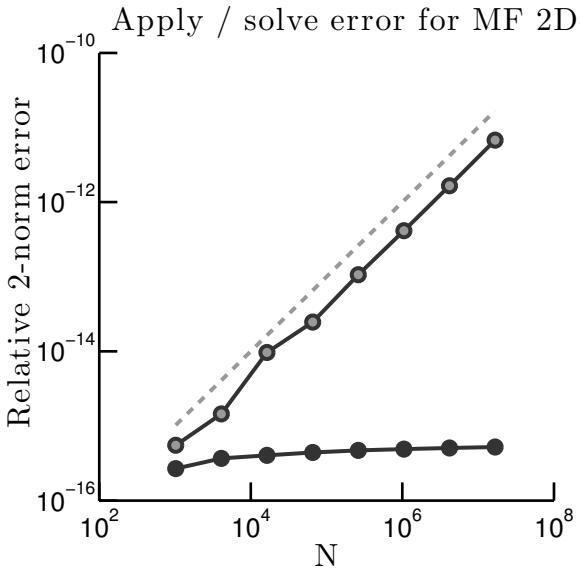

Figure 3: Relative error results for Example 1 in 2D. Black circles correspond to apply error for the factorization F, *i.e.*, $\|Fx - Ax\|_2/\|Ax\|_2$. Grey circles correspond to solve error, *i.e.*, $\|F^{-1}b - x\|_2/\|x\|_2$, where $Ax = b$. The guideline is $\mathcal{O}(N)$.

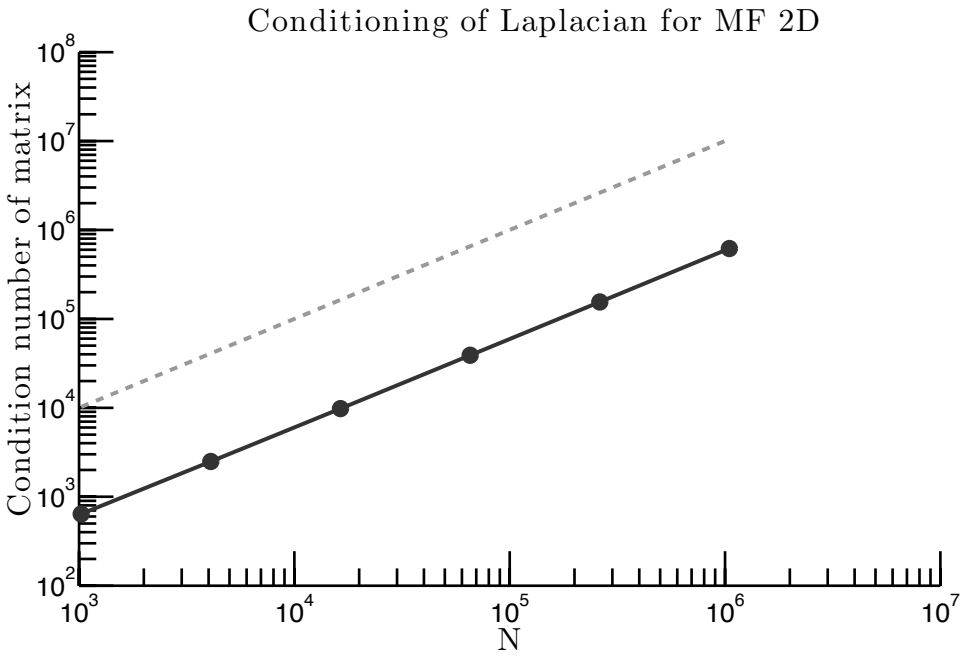

Figure 4: Conditioning of the example 1 system matrix in 2D as a function of N. The trendline is $\mathcal{O}(N)$.

Moving past the 2D examples, we turn to 3D scaling tests on the analogous problem, *i.e.*, we simply extend the grid to a 3D domain but keep the same material parameters. Figure 5 shows the timing results again for increased N, and we see that the factor time seems to agree with the $\mathcal{O}(N^2)$ trend line, with perhaps a slight uptick at the end. The apply / solve times stay as expected following the $\mathcal{O}(N^{4/3})$ curve. Investigating the error plots, we see in Figure 6 that the apply and solve error scales once again as expected relative to the conditioning estimate, which agrees with the theoretical conditioning of the matrix.

It is worth noting that in all cases, we also timed the relative amount of time spent performing linear algebra operations versus the amount of time spent doing other things, and observed that in all cases the time spent in linear algebra was about an order of magnitude larger than the time spent in copying, destructing, index manipulation, *etc.*

3.2 Example 2: Updating

As mentioned in the previous section, this code will go on to form the base of the updating code for sparse HIF and as such has updating functionality even for multifrontal, where updating should not be theoretically efficient. To investigate just how the practical scaling of updating might look for multifrontal, we constructed an updating test as follows.

Given the same initial problem as in example 1, construct a factorization. Then, in 2D we modify a small region of 60 points in one corner of the domain by doubling the parameter a at those points, with the number of modified points fixed at this number even for varying total number of points. Then, we time the amount of time it takes to follow the paths up the elimination tree corresponding to these modifed DOFs and refactor the corresponding blocks of the matrix.

We begin by looking at the 2D results, where we looked at how things scale not only for a standard 5-point Laplacian stencil ($w = 1$) but also a 9-point stencil ($w = 2$). These results can be seen in Figure 8. For the case $w = 1$, we see that we actually get a significant speedup for problems with N as big as 2048^2 (the largest data point), though it is clear that the speedup is diminishing for larger N. Even for the case $w = 2$, where the separators are twice as big, we see a speed up of between 5 and 10 for large problems, which is roughly around the boundary of "worth doing in practice" and "not".

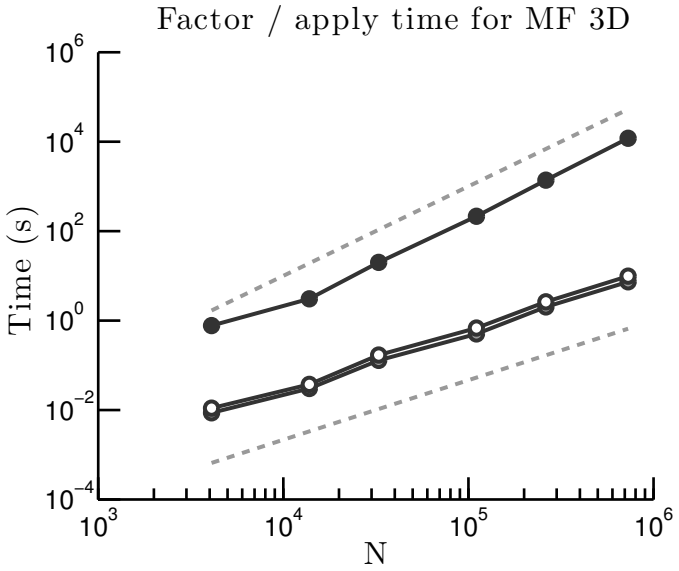

Figure 5: Timing results for Example 1 in 3D. Black circles correspond to factor time, white circles correspond to apply time, and grey circles correspond to solve time. The top guideline is $\mathcal{O}(N^2)$ and the bottom guideline is $\mathcal{O}(N^{3/2})$. Note that the apply and solve curves lay almost on top of each other.

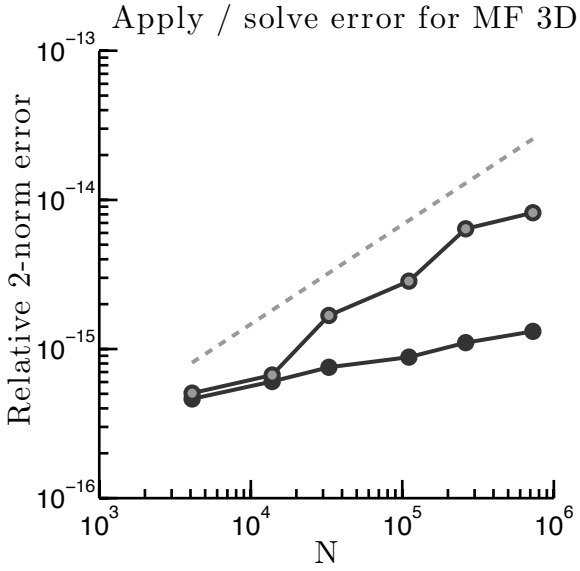

Figure 6: Relative error results for Example 1 in 3D. Black circles correspond to apply error for the factorization F, *i.e.*, $\|Fx - Ax\|_2/\|Ax\|_2$. Grey circles correspond to solve error, *i.e.*, $\|F^{-1}b - x\|_2/\|x\|_2$, where $Ax = b$. The guideline is $\mathcal{O}(N^{2/3})$.

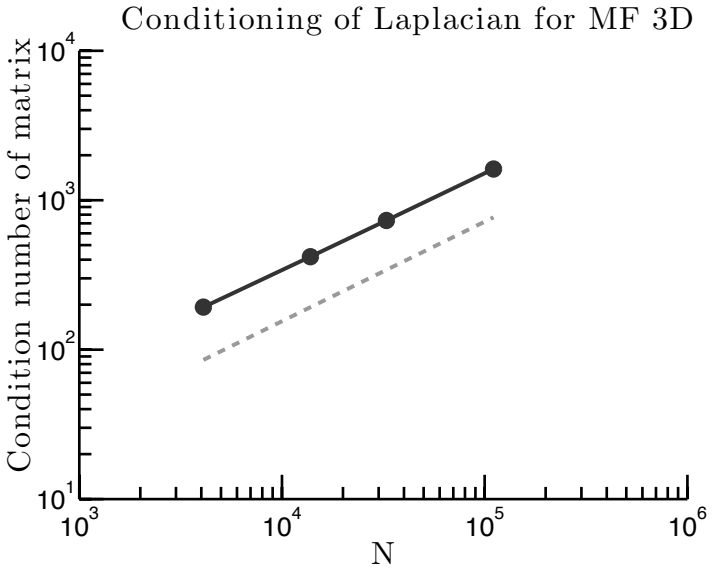

Figure 7: Conditioning of the example 1 system matrix in 3D as a function of N. The trendline is $\mathcal{O}(N^{2/3})$.

In 3D, we performed the same test for just a standard $w = 1$ Laplacian stencil, modifying a region of about 50 points as the total number of points varies. We see that, as expected, the efficiency is much worse in this case – we quickly move to the asymptotic regime where updating is essentially as expensive as the complete factorization. This is due to the fact that the cost of the root node is now even larger than in the 2D case – $\mathcal{O}(N^2)$ – and thus it dominates the cost of the factorization for smaller N, putting us in the regime of "this is probably not worth doing in practice." Of course, we still get a speedup of somewhere around 3X, which is a little surprising. These results can be seen in Figure 9.

It's worth noting that in both these examples, the modification is such that without updating or refactoring (*i.e.*, just using the incorrect old factorization and accepting the error) we get no more than 4 digits of accuracy in the solution when applying, as compared to 15 with the correct updated factorization. That is to say – these modifications are not trivial.

3.3 Example 3: Advection-diffusion

Consider the advection-diffusion equation (4). Because we are just interested in the linear system and not solving a particular problem, we will make the simplifying assumption that the velocity, material parameters, and source are constant with respect to time and the velocity is constant with respect to space so that we can consider the steady-state problem,

$$-\nabla \cdot (a(x)\nabla u(x)) + \mathbf{v} \cdot \nabla u(x) = f(x), \quad x \in \Omega. \tag{5}$$

For our example problem of interest, we will choose $\mathbf{v}$ to be positive in every component, such that upwinding always corresponds to a backward spatial difference. Note that this is just to simplify notation here for our specific problem and there is no problem extending beyond this assumption. Discretization of (5) in 2D with a Laplacian of width 1 and upwinding of the advection term leads to

$$\left(\frac{a_{i-1/2,j} + a_{i+1/2,j}}{h_{x_1}^2} + \frac{a_{i,j-1/2} + a_{i,j+1/2}}{h_{x_2}^2}\right) u_{ij} - \frac{a_{i-1/2,j}u_{i-1,j} + a_{i+1/2,j}u_{i+1,j}}{h_{x_1}^2}$$
$$- \frac{a_{i,j-1/2}u_{i,j-1} + a_{i,j+1/2}u_{i,j+1}}{h_{x_2}^2} + \frac{v_{x_1}(u_{i,j} - u_{i-1,j})}{h_{x_1}} + \frac{v_{x_2}(u_{i,j} - u_{i,j-1})}{h_{x_2}} = f_j$$

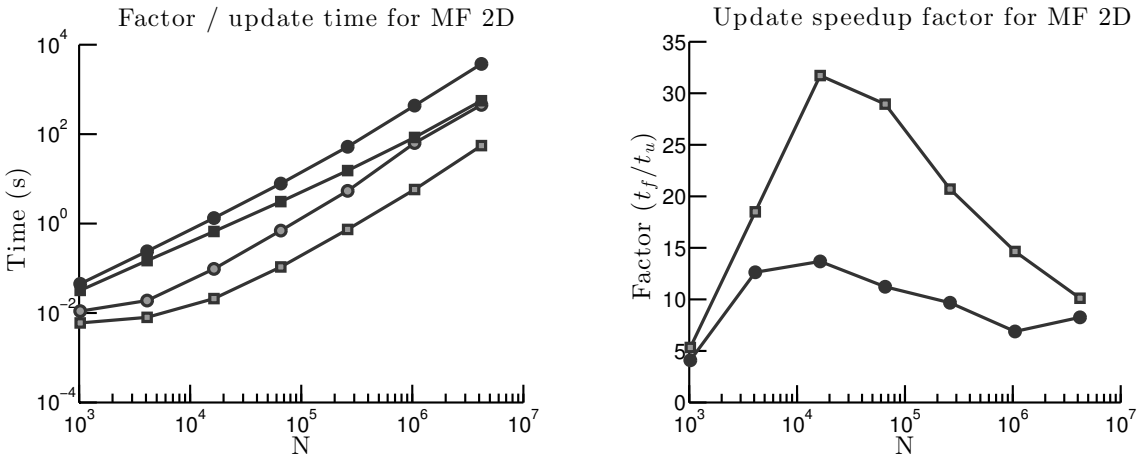

Figure 8: Left: The relative runtime for 2D factoring (black markers) versus updating (grey markers) as a function of N. The squares correspond to a separator width of $w = 1$ and the circles correspond to $w = 2$. Right: the speedup ratio given by factor time over update time for the cases $w = 1$ (grey) and $w = 2$ (black), corresponding to the same data as the left figure.

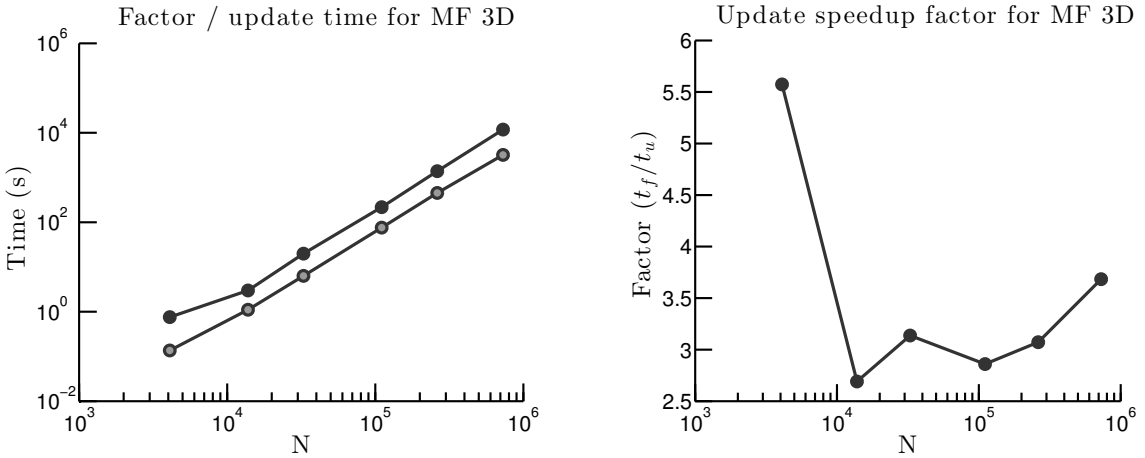

Figure 9: Left: The relative runtime for 3D factoring (black markers) versus updating (grey markers) as a function of N with width parameter $w = 1$. Right: the speedup ratio given by factor time over update time, corresponding to the same data as the left figure.

Table 1: Errors for example 3

N	Apply error (relative L2)	Solve error (relative L2)
128^2	5.54582e-16	1.08961e-15
256^2	5.6935e-16	2.50752e-15
512^2	5.74042e-16	3.64338e-15

So, given this, our example is as follows. We discretize the unit square with N_x DOFs in the x direction and N_y in the y direction with Dirichlet boundary conditions set to zero. We construct a random background conductivity field $a(x)$ with $a(x)$ distributed uniformly between $1e-10$ and 1. The velocity $\mathbf{v}$ is chosen such that $v_x = 100$ and $v_y = 20$. This puts us in roughly the "medium-sized Peclet number" regime, where both diffusion and advection are important. Then, we look at the apply and solve error for random vectors as well as the solution for a point source placed near the center of the domain.

For this example, we are interested in whether the method seems like it works or not for the advection matrices, *i.e.*, whether the algorithm will fail. Table 1 shows the relative and solve errors for some of the problem sizes we tested, and the corresponding solution for the source described above can be seen in Figure 10. We discuss these results in the discussion section.

Further

4 Discussion & Conclusion

For example 1 we note that, compared to the 2D scaling results for essentially the same problem on the same machine by Ho in [2], our factorization takes roughly a factor of 2 longer. We remind the reader that our implementation does not take advantage of symmetry, which may partially account for this, but also note that our design decisions do add some extra data structure overhead. Our implementation is in C++ whereas Ho's is in Matlab, so it is possible that this is due to the tuning of the Intel MKL. It is interesting to see as well that the scaling of complexity behaved just as expected. At first, the growth of solve error with N was concerning, but after investigating the condition number of the matrix A it became clear that the error can be explained as simply limits of floating point accuracy that we hit due to ill-conditioning of A.

For example 2, we are interested in the relative speed-up that might be attained by updating a multifrontal factorization rather than completely constructing a new factorization. This behaved better than expected. Asymptotically, we know that the scaling for updating should asymptotically approach zero benefit, but in practice we see nontrivial speedups in 2D and even in some of the 3D code. We note that the 3D results are a little less uniform than the 2D results due to the non-power-of-two problem sizes tested. It seems that in practice, updating could be a useful technique for these multifrontal techniques if one is in the regime where localized updates make sense as a paradigm. In future work, we will explore just how much more of a speedup we can attain by updating HIF instead of multifrontal, taking advantage of the compression of the fronts performed in the HIF algorithm.

For example 3, the basic experimental result is that the algorithm did not break, which is perhaps interesting in itself. However, it is not altogether surprising. If we look at the stencil of the advection-diffusion equation we solve, we see that the system matrix is both row- and column-diagonally dominant, though not necessarily strictly. However, if we view the system matrix as $A = L+V$, where L is the Laplacian from the diffusion term and V is the matrix of advection terms, we see that L is positive definite and V is positive semidefinite, so we start to see why it might be the case that we could construct a triangular factorization of A without pivoting. Obviously, our tests are only preliminary, but it is worth noting that we tried a number of velocity ranges, size ranges, and coefficient fields, and were unable to break the algorithm.

If you have any questions / comments / concerns, feel free to email me.

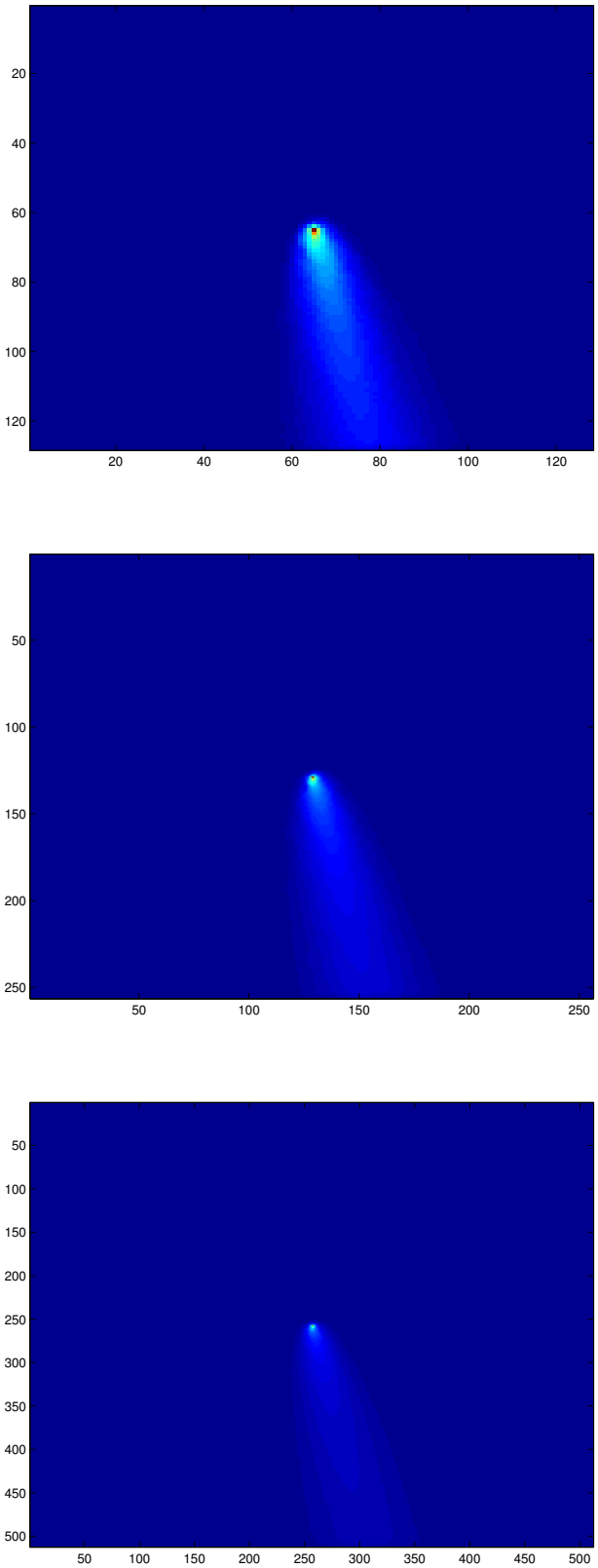

Figure 10: The solution profile for example 3 with $N = 128^2, 256^2, 512^2$, from top to bottom.

References

[1] A. George, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.

[2] K. Ho and L. Ying, *Hierarchical interpolative factorization for elliptic operators: differential equations*, ArXiv e-prints, (2013).

[3] J. W. H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Review, 34 (1992), pp. 82–109.

[4] V. Minden, A. Damle, K. L. Ho, and L. Ying, *A technique for updating hierarchical factorizations of integral operators*, ArXiv e-prints, (2014).