Report for 2019 applicants by Peiqi Wang

This report explains my efforts in trying to implement forward substitution method on sparse matrices. Sequantially, I made the following implementations here

1. (`src/formats.h`) I/O code for interfacing with Matrix Market format files
2. (`tests.cpp`) tests that verifies the correctness of the method
3. (`src/triangular.h`) forward substitution method that relies on finding the reachset
4. (`src/triangular.h`) a method that relies on `OpenMP` naively
5. (`scripts/codegen.ipynb`) code generation routine that takes into account sparsity patterns

1 The Problem

Find efficient methods to solve $\mathbf{Lx} = \mathbf{b}$ where both $\mathbf{L}, \mathbf{b}$ are sparse and $\mathbf{L}$ is lower triangular

2 Verification Strategies

I implemented two verification strategies (`tests.cpp#L126`)

1. (`tests.cpp#L136`) Given a tentative solution $\mathbf{x}$, compute $\mathbf{Lx}$ and show $\mathbf{Lx} \approx \mathbf{b}$
2. (`scripts/triangular.m#L23`) Compute $\mathbf{L}\backslash\mathbf{b}$ in MATLAB (and Julia) and show $\mathbf{x} \approx \mathbf{L}\backslash\mathbf{b}$

I noticed that strategy (1) show agreement for small matrices but indicates that $\mathbf{x}$ is computed incorrectly for large matrices. After some debugging, I found that the matrix accumulates error and are ill-conditioned (verified with `scripts/cond.m`). During testing, (1) is tested against small matrices while (2) is tested against all available matrices (that I tried).

3 Code Generation

Two purpose of code generation in this specific problem

1. reduce array access since position of nonzero values in $\mathbf{L}, \mathbf{b}$ are known beforehand
2. allow for low level optimizations such as loop peeling, vectorization, etc.

I code up a script to generate some code for `torso` and `tsopf`. However I only managed to achieve goal (1) and partially goal (2). The code never finishes compilation when `-O3` is set so goal (2) is only partially achieved by manual loop peeling during the code generation process.

4 Methods and Results

Several variations of forward substitution are compared

1. (`lsolve_simple`) does not consider sparsity
2. (`lsolve_eigen`) updates to nonzero values in j-th column of $\mathbf{L}$ is skipped if $\mathbf{x}_k$ is 0
3. (`lsolve_reachset_default`)
   - (*symbolic*) computes $\texttt{Reach}_L(\mathbf{b}) = \{j \mid \mathbf{x}_j \neq 0\}$ using depth-first-search
   - (*numeric*) performs numerical solve in topological order of $\texttt{Reach}_L(\mathbf{b})$
4. (`lsolve_reachset_{small,medium,torso,tsopf}`) are variants of (3); code is generated with information of sparsity pattern of matrix. For now, the generated code has loop peeled manually, by referencin the Sympiler paper. Vectorization, tiling, etc. are not implemented as compiler directives as I am not too familiar with them
5. (`lsolve_eigen_par`) has inner loop naively wrapped in a `#pragma omp parallel for`

Performance of above methods are compared by running each method 10 times, whereby average is taken and normalized to that of `solve_simple`. Two sparse matrix lhs are used as inputs, specifically torso and tsopf. Code is compiled on MacOS 10.13.6 with Clang with `-O0`. As mentioned previously, it takes too long to compile code with `-O3` since file size for generated files is too large, i.e. `src/codegen_torso.cpp` is approximately `3.5MB`.
From Figure 1, we see

1. significant performance gain for all other methods from using reachset for `tsopf`, and moderate performance for `lsolve_eigen` and the reachset based methods. This is due to the fact that `torso` has a lot more nonzero elements; both numeric and symbolic time for sparse matrix increases with the number of nonzero elements.
2. `lsolve_eigen` is simple to implement and performs quite well in these two cases. This is due to the fact that certain loops are skipped and runtime $O(n) \rightarrow O(nnz)$
3. `lsolve_reachset_torso` has a slightly better numeric runtime performance when compared to `lsolve_reachset_default`, this is mostly due to the fact that code generation removes quite a number of array access operations. The code could get even faster if vectorization, tiling and other optimization methods are enabled.
4. `lsolve_eigen_par` is my only attempt to parallelize the code and a bad one. The performance is worse due to extra cost of thread creation, synchronization, and communication. Additionally, threads do not have enough work to do before they are synchronized to enter the next outer loop. Something I would probably have tried is something that ParSy have come up with, i.e. coarse level parallelism to decrease synchronization cost while preserving data dependency

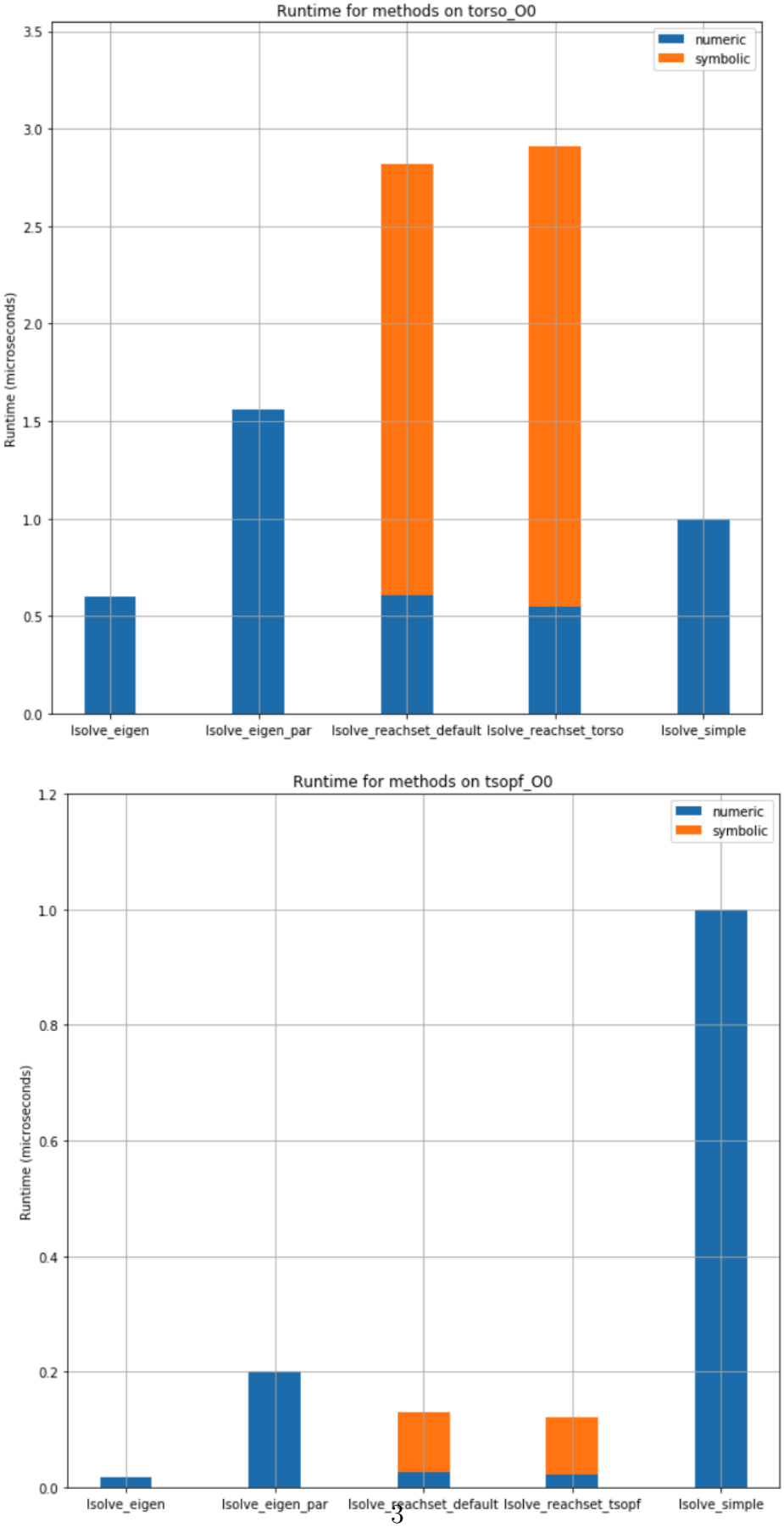

Figure 1: runtime on `torso` and `tsopf` with `-O0`