What "Provably Fast" Can Mean in Practice: A Case Study with a Combinatorial Laplacian Solver*

Daniel Hoske[1,2] Dimitar Lukarski[3] Michael Wegner[1]
Henning Meyerhenke[1,4]

[1] Faculty of Informatics, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, D-76131 Karlsruhe, Germany
[2] Google Inc., Mountain View, CA, USA
[3] Paralution Labs UG & Co. KG, Gaggenau, Germany
[4] Corresponding author, email: meyerhenke@kit.edu, phone: +49-721-608-41876, fax: +49-721-608-44211

Abstract

Linear system solving is a main workhorse in applied mathematics. Recently, theoretical computer scientists contributed sophisticated algorithms for solving linear systems with symmetric diagonally dominant (SDD) matrices in provably nearly-linear time. These algorithms are very interesting from a theoretical perspective, but their practical performance was unclear.

Here we address this gap. We provide the first implementation of the combinatorial solver by Kelner et al. [STOC 2013], which is appealing for implementation due to its conceptual simplicity. The algorithm exploits that a Laplacian matrix (which is SDD) corresponds to a graph; solving symmetric Laplacian linear systems amounts to finding an electrical flow in this graph with the help of cycles induced by a spanning tree with the low-stretch property.

The results of our experiments are ambivalent. While they confirm the predicted nearly-linear running time, the constant factors make the solver much slower for reasonable inputs than basic methods with higher asymptotic complexity. We were also not able to use the solver effectively as smoother or preconditioner. Moreover, while spanning trees with lower stretch indeed reduce the solver's running time, we experience again a discrepancy in practice: In our experiments simple spanning tree algorithms perform better than those with a guaranteed low stretch.

1 Introduction

Solving square linear systems $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$, is one of the most important problems in applied mathematics with wide applications in science and engineering. In practice system matrices are often *sparse*, i. e. they contain $O(n)$ nonzeros. Ideally, the required time for solving sparse systems would grow linearly with the number of nonzeros $2m$. Most direct solvers, however, show cubic running times and do not exploit sparsity. Also, approximate solutions usually

*Parts of this paper have been published in a preliminary form in the Proceedings of the 14th International Symposium on Experimental Algorithms (SEA 2015) [17]. Most work was done while D. Hoske was with KIT. This work was partially supported by the Ministry of Science, Research and the Arts Baden-Württemberg.

suffice due to the imprecision of floating point arithmetic. Exploiting this fact with sparse iterative solvers such as conjugate gradient (CG) still yields a running time that is clearly superlinear in n.

Spielman and Teng [31], following an approach proposed by Vaidya [34], achieved a major breakthrough in this direction by devising a nearly-linear time algorithm for solving linear systems in symmetric diagonally dominant matrices. *Nearly-linear* means $\mathcal{O}\big(m \cdot \text{polylog}(n) \cdot \log(1/\epsilon)\big)$ here, where $\text{polylog}(n)$ is the set of real polynomials in $\log(n)$ and ϵ is the relative error $\|x - x_{\text{opt}}\|_A / \|x_{\text{opt}}\|_A$ we want for the solution $x \in \mathbb{R}^n$. Here $\|\cdot\|_A$ is the norm $\|x\|_A := \sqrt{x^T A x}$ given by A, and $x_{\text{opt}} := A^+ b$ is an exact solution (where A^+ refers to the pseudoinverse of A). A matrix $A = (a_{ij})_{i,j \in [n]} \in \mathbb{R}^{n \times n}$ is *diagonally dominant* if $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$ for all $i \in [n]$.

Symmetric matrices that are diagonally dominant (SDD matrices) have many applications, e.g. in elliptic PDEs [7], maximum flows [11], and sparsifying graphs [30]; also see [18]. Thus, the problem INV-SDD of solving linear systems $Ax = b$ for x on SDD matrices A is of significant importance. We focus here on Laplacian matrices (which are SDD) due to their rich applications in algorithms for undirected graphs, e. g. load balancing [13, 25], but this is no major limitation [19].

Related work. Spielman and Teng's seminal paper [31] requires a lot of sophisticated machinery: a multilevel approach [34, 28] using recursive preconditioning, preconditioners based on low-stretch spanning trees [32] and spectral graph sparsifiers [30, 21]. Later papers extended this approach, both by making it simpler and by reducing the exponents of the polylogarithmic time factors.[1] We focus on a simplified algorithm by Kelner et al. [19] that reinterprets the problem of solving an SDD linear system as finding an electrical flow in a graph. It only needs low-stretch spanning trees and achieves $\mathcal{O}\big(m \log^2 n \log\log n \log(1/\epsilon)\big)$ time. Another interesting nearly-linear time SDD solver is the recursive sparsification approach by Peng and Spielman [27]. Together with a parallel sparsification algorithm (e.g. [20]) it yields a nearly-linear work parallel algorithm.

Spielman and Teng's algorithm crucially uses the low-stretch spanning trees first introduced by Alon et al. [3]. (For a definition of stretch see Section 2.) Elkin et al. [14] provide an algorithm for computing spanning trees with polynomial stretch in nearly-linear time. Specifically, they get a spanning tree with $\mathcal{O}(m \log^2 n \log\log n)$ stretch in $O(m \log^2 n)$ time. Abraham et al. [1, 2] later showed how to get rid of some of the logarithmic factors in both stretch and time. Papp [26] tested these algorithms in practice and showed that they do not usually result in spanning trees with lower stretch than a simple minimum-weight spanning tree computed with Kruskal's algorithm [22] and that Elkin et al.'s original algorithm [14] achieves the best results among the provably good approaches. We use these low-stretch spanning trees in our implementation of Kelner et al's. [19] algorithm and compare their effectiveness for the solver.

Motivation, Outline, and Contribution. Although several extensions and simplifications to Spielman and Teng's nearly-linear time solver [31] have been proposed, there is a lack of results how they all perform in practice. We seek to fill this gap by implementing and evaluating an algorithm proposed by Kelner et al. [19] that is fascinating due to its simple description and easier to implement (and thus more promising in practice) than the original Spielman-Teng algorithm.

Hence, in this paper we implement the KOSZ solver (the acronym follows from the authors' last names) by Kelner et al. [19] and investigate its practical performance. To this end, we start

[1] Spielman provides a comprehensive overview of later work at https://sites.google.com/a/yale.edu/laplacian/ (accessed on November 18, 2015).

in Section 2 by describing important notation and outlining KOSZ. In Section 3 we elaborate on the design choices one can make when implementing KOSZ. In particular, we explain when these choices result in a provably nearly-linear time algorithm. Section 4 contains the experimental evaluation of the Laplacian solver KOSZ. We consider the configuration options of the algorithm, its asymptotics, its convergence and its use as a preconditioner or as a smoother. Our results confirm a nearly-linear running time, but are otherwise disappointing from a practical point of view: The asymptotics hide very high constant factors, in part due to memory accesses. We conclude the paper in Section 5 by summarizing the results and discussing future research directions.

2 Preliminaries

Fundamentals. We consider undirected simple graphs $G = (V, E)$ with n vertices and m edges. A graph is *weighted* if we have an additional function $w\colon E \to \mathbb{R}_{>0}$. Where necessary we consider unweighted graphs to be weighted with $w_e = 1\ \forall e \in E$. We usually write an edge $\{u, v\} \in E$ as uv and its weight as w_{uv}. Moreover, we define the set operations $\cup$, $\cap$ and $\setminus$ on graphs by applying them to the set of vertices and the set of edges separately. For every node $u \in V$ its *neighborhood* $N_G(u)$ is the set $N_G(u) := \{v \in V : uv \in E\}$ of vertices v with an edge to u and its *degree* d_u is $d_u = \sum_{v \in N_G(u)} w_{uv}$. The *Laplacian matrix* of a graph $G = (V, E)$ is defined as $L_{u,v} := -w_{uv}$ if $uv \in E$, $\sum_{x \in N_G(u)} w_{ux}$ if $u = v$, and 0 otherwise for $u, v \in V$. A Laplacian matrix is always an SDD matrix. Another useful property of the Laplacian is the factorization $L = B^T R^{-1} B$, where $B \in \mathbb{R}^{E \times V}$ is the *incidence matrix* and $R \in \mathbb{R}^{E \times E}$ is the *resistance matrix* defined by $B_{ab,c} = 1$ if $a = c$, $= -1$ if $b = c$, and 0 otherwise. $R_{e_1,e_2} = 1/w_{e_1}$ if $e_1 = e_2$ and 0 otherwise. This holds for all $e_1, e_2 \in E$ and $a, b, c \in V$, where we arbitrarily fix a start and end node for each edge when defining B. With $x^T L x = (Bx)^T R^{-1} (Bx) = \sum_{e \in E} (Bx)_e^2 \cdot w_e \geq 0$ (every summand is non-negative), one can see that L is *positive semidefinite*. (A matrix $A \in \mathbb{R}^{n \times n}$ is positive semidefinite if $x^T A x \geq 0$ for all $x \in \mathbb{R}^n$.)

Cycles, Spanning Trees, and Stretch. A *cycle* in a graph is usually defined as a simple path that returns to its starting point and a graph is called *Eulerian* if there is a cycle that visits every edge exactly once. In this work we will interpret cycles somewhat differently: We say that a cycle in G is a subgraph C of G such that every vertex in G is incident to an even number of edges in C, i. e. a cycle is a union of Eulerian graphs. It is useful to define the addition $C_1 \oplus C_2$ of two cycles C_1, C_2 to be the set of edges that occur in exactly one of the two cycles, i. e. $C_1 \oplus C_2 := (C_1 \setminus C_2) \cup (C_2 \setminus C_1)$. In algebraic terms we can regard a cycle as a vector $C \subseteq \mathbb{F}_2^E$ ($\mathbb{F}_2$ is the finite field of order 2) such that $\sum_{v \in N_C(u)} 1 = 0$ in $\mathbb{F}_2$ for all $u \in V$ and the cycle addition as the usual addition on $\mathbb{F}_2^E$. We call the resulting linear space of cycles $\mathcal{C}(G)$.

In a *spanning tree* (ST) $T = (V, E_T)$ of G there is a unique path $P_T(u, v)$ from every node u to every node v. For any edge $e = uv \in E \setminus E_T$ (an *off-tree-edge with respect to* T), the subgraph $e \cup P_T(u, v)$ is a cycle, the *basis cycle induced by* e. One can easily show that the basis cycles form a basis of $\mathcal{C}(G)$. Thus, the basis cycles are very useful in algorithms that need to consider all the cycles of a graph. Another notion we need is a measure of how well a spanning tree approximates the original graph. We capture this by the *stretch* $\mathrm{st}(e) = \left(\sum_{e' \in P_T(u,v)} w_{e'}\right)/w_e$ *of an edge* $e = uv \in E$. This stretch is the detour you need in order to get from one endpoint of the edge to the other if you stay in T, compared to the length of the original edge. In the literature the stretch

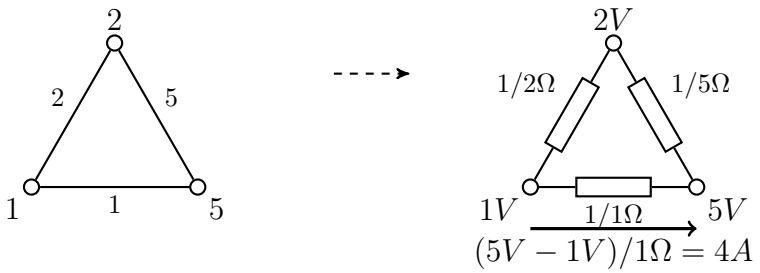


Figure 1: Transformation into an electrical network.

is sometimes defined slightly differently, but we follow the definition in [19] using w_e. The *total stretch* of the whole tree T is the sum of the individual stretches $\mathrm{st}(T) = \sum_{e \in E} \mathrm{st}(e)$. Finally, we define the average stretch as the total stretch divided by the total edge weight. Finding a spanning tree with low stretch is crucial for proving the fast convergence of the KOSZ solver.

Electrical Network Analogy. We can regard G as an electrical network where each edge uv corresponds to a resistor with conductance w_{uv} and x as an assignment of potentials to the nodes of G (cf. Figure 1). L operates on every vector $x \in \mathbb{R}^n$ via $(Lx)_u = \sum_{v \in N(u)} (x_u - x_v) \cdot w_{uv}$ for each $u \in V$. Then $x_u - x_v$ is the voltage across uv and $(x_u - x_v) \cdot w_{uv}$ is the resulting current along uv. Thus, $(Lx)_u$ is the current flowing out of u that we want to be equal to the right-hand side b_u. Furthermore, one can reduce solving SDD systems to the related problem INV-LAPLACIAN-CURRENT [19]: Given a Laplacian $L = L(G)$ and a vector $b \in \mathrm{im}(L)$, compute a function $f \colon \widetilde{E} \to \mathbb{R}$ with (i) f being a valid graph flow on G with demand b and (ii) the potential drop along every cycle in G being zero, where a valid graph flow means that the sum of the incoming and outgoing flow at each vertex respects the demand in b and that $f(u, v) = -f(v, u)\ \forall uv \in E$. Also, $\widetilde{E}$ is a bidirected copy of E and the potential drop of cycle C is $\sum_{e \in C} f(e) r_e$.

KOSZ (Simple) Solver. The idea of the algorithm is to start with any valid flow and successively adjust the flow such that every cycle has potential zero. We need to transform the flow back to potentials at the end, but this can be done consistently, as all potential drops along cycles are zero.

Regarding the crucial question of what flow to start with and how to choose the cycle to be repaired in each iteration, Kelner et al. [19] suggest using the cycle basis induced by a spanning

Input: Laplacian $L = L(G)$ and vector $b \in \mathrm{im}(L)$.
Output: Solution x to $Lx = b$.
$T \leftarrow$ a spanning tree of G
$f \leftarrow$ unique flow with demand b that is only nonzero on T
while *there is a cycle with potential drop* $\neq 0$ *in* f **do**
  $c \leftarrow$ cycle in $\mathcal{C}(T)$ chosen randomly weighted by stretch
  $f \leftarrow f - \frac{c^T R f}{c^T R c} c$
return *vector of potentials in* f *with respect to the root of* T

Algorithm 1: INV-LAPLACIAN-CURRENT solver KOSZ.

tree T of G and prove that the convergence of the resulting solver depends on the stretch of T. More specifically, they suggest starting with a flow that is nonzero only on T and weighting the basis cycles proportionate to their stretch when sampling them. The resulting algorithm is shown as Algorithm 1; note that we may stop before all potential drops along cycles are zero and we can consistently compute the *potentials induced by* f at the end by only looking at T.

The solver described in Algorithm 1 is actually just the `SimpleSolver` in Kelner et al.'s [19] paper. They also show how to improve this solver by adapting preconditioning to the setting of electrical flows. In informal experiments we could not determine a strategy that is consistently better than the `SimpleSolver`, so we do not pursue this scheme any further here. Eventually, Kelner et al. derive the following running time for the KOSZ (simple) solver:

Theorem 1. *[19, Thm. 3.2]* `SimpleSolver` *can be implemented to run in time* $O(m \log^2 n \log\log n \log(\epsilon^{-1} n))$ *for computing an* ϵ*-approximation of* x.

3 Implementation

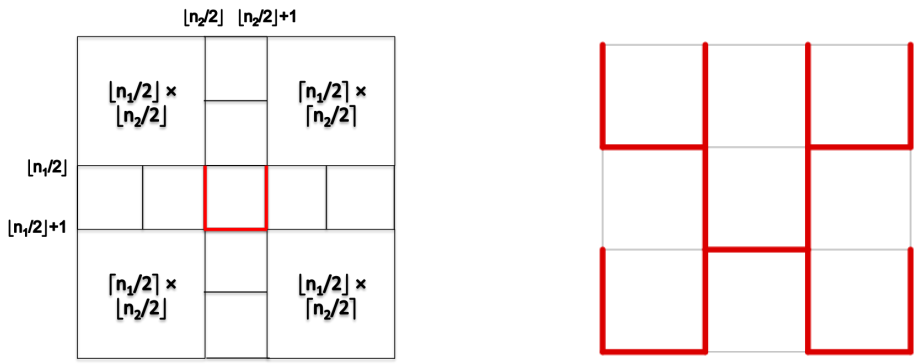
While Algorithm 1 provides the basic idea of the KOSZ solver, it leaves open several implementation decisions that we elaborate on in this section.

3.1 Spanning trees

As suggested by the convergence result in Theorem 1, the KOSZ solver depends on low-stretch spanning trees. The notion of stretch was introduced by Alon et al. [3] along with an algorithm to compute a spanning tree with low stretch. Unfortunately, the stretch guaranteed by their algorithm is super-polynomial. Elkin et al. [14] presented an algorithm requiring nearly-linear time and yielding nearly-linear average stretch. The basic idea is to recursively form a spanning tree using a star of balls in each recursion step. We use Dijkstra with binary heaps for growing the balls and take care not to need more work than necessary to grow the ball. In particular, ball growing is output-sensitive and growing a ball $B(x, r) := \{v \in V : \text{distance from } x \text{ to } v \text{ is} \leq r\}$ should require $\mathcal{O}(d \log n)$ time where d is the sum of the degrees of the nodes in $B(x, r)$. The exponents of the logarithmic factors of the stretch of this algorithm were improved by subsequent papers, but Papp [26] showed experimentally that these improvements do not yield better stretch in practice. In fact, his experiments suggest that the stretch of the provably good algorithms is usually not better than just taking a minimum-weight spanning tree. Therefore, we additionally use two simpler spanning trees without stretch guarantees: A minimum-distance spanning tree with Dijkstra's algorithm (the tree built implicitly during the search) and binary heaps; as well as a minimum-weight spanning tree with Kruskal's algorithm using union-find with union-by-size and path compression.

To test how dependent the algorithm is on the stretch of the ST, we also look at a *special ST* for $n_1 \times n_2$ grids. As depicted in Figure 2, we construct this spanning tree by subdividing the $n_1 \times n_2$ grid into four subgrids as evenly as possible (the subgrid sizes are shown in Figure 2(a)), recursively building the STs in the subgrids (the termination of the recursion is shown in Figure 2(b)) and connecting the subgrids by a U-shape in the middle.

Proposition 2. *Let* G *be an* $n_1 \times n_2$ *grid with* $n_1, n_2 \geq 4$. *Then the special ST has* $\mathcal{O}\left(\frac{(n_1+n_2)^2 \log(n_1+n_2)}{n_1 n_2}\right)$ *average stretch on* G.



(a) Recursive construction

(b) ST for $n_1 = n_2 = 4$

Figure 2: Special spanning tree with $\mathcal{O}\big(\frac{(n_1+n_2)^2 \log(n_1+n_2)}{n_1 n_2}\big)$ average stretch for the $n_1 \times n_2$ grid.

Proof. First note that, by the recursive construction, the total stretch of the four subgrids remains the same if such a subgrid is treated separately. Moreover, the stretches of the $\mathcal{O}(n_1 + n_2)$ off-tree edges between the rows $\lfloor n_1/2 \rfloor$ and $\lfloor n_1/2 \rfloor + 1$ as well as the columns $\lfloor n_2/2 \rfloor$ and $\lfloor n_2/2 \rfloor + 1$ are in $\mathcal{O}(n_1 + n_2)$ each. To see this, let s and t be the source and target vertices of such an off-tree edge, respectively. Then, by construction, it is possible to reach the center from s in $\mathcal{O}(n_1 + n_2)$ steps and t from the center likewise. Consequently, $S\big(n_1, n_2\big) = 4 \cdot S\big(n_1/2, n_2/2\big) + \mathcal{O}\big(n_1 + n_2\big)^2$ when disregarding rounding. After solving this recurrence (note that $S(n_1/2, n_2/2)$ is essentially one fourth in size compared to $S(n_1, n_2)$ as long as $n_1, n_2 \geq 4$), we get

$$S\big(n_1, n_2\big) = \mathcal{O}\big((n_1 + n_2)^2 \log(n_1 + n_2)\big).$$

Since the number of edges is $\Theta(n_1 n_2)$, the claim for the average stretch follows. □

In case of a square grid ($n_1 = n_2$) with $N = n_1 \times n_2$ vertices, we get $S(N) = 4S(N/4) + \mathcal{O}(N) = \mathcal{O}(N \log N) = \mathcal{O}(n_1^2 \log(n_1))$ and thus $\mathcal{O}(\log n_1)$ average stretch. A logarithmic average stretch (and thus detour) is noteworthy since the average distance between a random pair of nodes in the square grid is $\Omega(n_1)$. Also, for this special case, our result slightly improves on the general low-stretch spanning tree algorithms. Later on in this paper, we will use it in comparison to other spanning trees to assess their effect on the KOSZ solver.

3.2 Flows on trees

Since every basis cycle contains exactly one off-tree-edge, the flows on off-tree-edges can simply be stored in a single vector. To be able to efficiently get the potential drop of every basis cycle and to be able to add a constant amount of flow to it, the core problem is to efficiently store and update flows in T. We want to support the following operations for all $u, v \in V$ and $\alpha \in \mathbb{R}$ on the flow f:

- query(u, v): return the potential drop $\sum_{e \in P_T(u,v)} f(e) r_e$
- update(u, v, α): set $f(e) := f(e) + \alpha$ for all $e \in P_T(u, v)$

} (1)

We can simplify the operations by fixing v to be the root r of T:

- $\mathrm{query}(u)$: return the potential drop $\sum_{e \in P_T(u,r)} f(e) r_e$ and
- $\mathrm{update}(u, \alpha)$: set $f(e) := f(e) + \alpha$ for all $e \in P_T(u, r)$.

} (2)

The itemized two-node operations can then be supported with $\mathrm{query}(u, v) := \mathrm{query}(u) - \mathrm{query}(v)$ and $\mathrm{update}(u, v, \alpha) := \big\{\mathrm{update}(u, \alpha) \text{ and } \mathrm{update}(v, -\alpha)\big\}$ since the changes on the subpath $P_T\big(r, \mathrm{LCA}(u, v)\big)$ cancel out. Here $\mathrm{LCA}(u, v)$ is the *lowest common ancestor* of the nodes u and v in T, the node farthest from r that is an ancestor of both u and v. We provide two approaches for implementing the operations; they are described next in some detail.

Linear time updates. The trivial implementation of (2) directly stores the flows in the tree and implements each operation in (2) with a single traversal from the node u to the root r. We can improve this implementation by only traversing up to $\mathrm{LCA}(u, v)$ in (1). Of course, this does not help with the worst-case time $\mathcal{O}(n)$, but could be quite significant in practice since basis cycles are often short. Data structures that answer LCA queries for pairs of nodes after some precomputation are a classical topic, optimal solutions are known [16, 6]. In our implementation we use a simpler implementation with higher (but still insignificant) preprocessing time that transforms an LCA query into a range minimum query (RMQ), the problem of determining the minimum in a subrange of an array. We can then solve the RMQ problem by precomputing the RMQ of every range whose length is a power of two, i. e. for each i with $2^i \leq n$ and every $j \in [n]$ we compute $\texttt{prec}[i, j] := \arg\min v[j \dots j + 2^i - 1]$. This can be done in $\mathcal{O}\big(n \log n\big)$ time.

Logarithmic time updates. While the data structure presented above allows fast repairs for short basis cycles, the worst-case time is still in $\mathcal{O}(n)$. We therefore also implement the data structure by Kelner et al. [19] with $\mathcal{O}(\log n)$ worst-case time repairs. It is based on link-cut trees [29]. The first observation it uses is that every rooted tree T on n nodes can be decomposed into edge-disjoint subtrees intersecting in exactly one node such that each subtree has $\leq n/2$ nodes. Equivalently, we find a vertex in T all of whose induced subtrees have size $\leq n/2$. We call such a vertex a *good vertex separator*. By recursively finding good vertex separators on the subtrees, we get a recursive decomposition of the whole tree into subtrees. Since the size of the trees halves in each step, the depth of this decomposition is at most $\mathcal{O}(\log n)$.

Remark 3. *We can implement* `query` *and* `update` *efficiently by storing several values: (i)* d_{drop}*: the total potential drop on the path* $P_T(r, d)$*, (ii)* d_{ext}*: the total flow contribution to* $P_T(r, d)$ *from vertices below* d*, and (iii)* $\mathrm{height}(u) := \sum_{e \in P_T(r,a) \cap R_T(r,d)} r_e$ *for every* $u \in V(T)$*, i. e. the accumulated resistance in common between the* $P_T(r, d)$ *path and the* $P_T(r, a)$ *path.*

Then we can compute $\mathrm{query}(u)$ as follows: If $u \in T_0$, the potential drop consists of the potential drop $\mathrm{query}_{T_0}(u)$ in T_0 and the part $d_{\text{ext}} \cdot \mathrm{height}(u)$ of the potential drop caused by vertices beyond d. If, however, $u \in T_i$ and $u \neq d$, then we have the complete potential drop d_{drop} along $P_T(d, r)$ and a recursive potential drop $\mathrm{query}_{T_i}(u)$.

The $\mathrm{update}(u, \alpha)$ operation can be implemented similarly: If $u \notin T_0$, we need to adjust d_{ext} by α. In all cases we need to update d_{drop} by the $\mathrm{height}(u)$ part of the $P_T(r, u)$ path in common with T_0. Unless $u = d$, we then need to recursively update the tree T_i with $u \in T_i$. While we could directly implement this recursion, we unroll it to get a more efficient implementation. We can store the complete state of the data structure in a dense vector x containing the d_{drop} and d_{ext}

values for all recursion levels. For each $u \in T$, query is then a dot product $q(u) \cdot x$ with a vector $q(u)$ containing the appropriate coefficients and $\text{update}(u, \alpha)$ is a vector addition $x := x + \alpha l(u)$ with a vector $l(u)$. The vectors $q(u)$ and $l(u)$ are sparse with at most $\mathcal{O}(\log n)$ nonzero entries and can be determined directly from the recursive decomposition in $\mathcal{O}\big(n \log(n)\big)$ time (their entries are either $\text{height}(u)$ or 1). Kelner et al. [19] provide more details about the unrolling.

Results. In our experiments (details omitted due to space constraints) the cost of querying the LCA-based data structure (LCAFlow) strongly depends on the structure of the used spanning tree, while the logarithmic-time data structure (LogFlow) induces costs that stay nearly the same. Similarly, the cost of LCAFlow grows far more with the size of the graph than LogFlow and LogFlow wins for the larger graphs in both classes. For these reasons, we only use LogFlow in later results.

3.3 Remarks on Initial Solution and Cycle selection

Given x we can compute a flow f via $f_{uv} := x(u) - x(v)$. The potential drop of each cycle in this flow f is zero. Unfortunately, this flow is not a valid graph flow with demand b – unless x already fulfills $Lx = b$. In contrast, in the solver we iteratively establish the zero-cycle-sum property from the flow originally induced by the spanning tree T. There is an important consequence: We cannot start from an arbitrary vector x, which may make it harder to use the solver in a larger context.

The easiest way to select a cycle, in turn, is to choose an off-tree edge *uniformly at random* in $\mathcal{O}(1)$ time. However, to get provably good results, we need to weight the off-tree-edges by their stretch, i. e. edges chosen with probability proportionate to their stretch. We can use the flow data structure described above to get the stretches. More specifically, the data structure initially represents $f = 0$. For every off-tree edge uv we first execute $\text{update}(u, v, 1)$, then $\text{query}(u, v)$ to get $\sum_{e \in P_T(u,v)} r_e$ and finally $\text{update}(u, v, -1)$ to return to $f = 0$. This results in $\mathcal{O}(m \log n)$ time to initialize cycle selection. Once we have the weights, we use *roulette wheel selection* in order to select a cycle in $\mathcal{O}(\log m)$ time after an additional $\mathcal{O}(m)$ time initialization. Roulette wheel selection is a simple strategy to sample an arbitrary discrete distribution with finite support: (i) Let X be a random variable with $\text{Prob}[X = x_i] = p_i$ for $i \in [k]$. (ii) Precompute the prefix sums $P = (0, p_1, p_1 + p_2, \ldots, p_1 + \cdots + p_k = 1)$. (iii) To sample, choose a uniform random value $x \in [0, 1)$. Then find the index i with $P_i \leq x < P_{i+1}$ using binary search and output x_i. The probability for getting x_i with this scheme is $\left| \left[\sum_{j=0}^{i-1} p_i, \sum_{j=0}^{i} p_i \right) \right| = p_i$, as desired.

4 Evaluation

4.1 Settings

Software, hardware, and data. We implemented the KOSZ solver in C++ using NetworKit [33], a tool suite focused on large-scale network analysis. Our code is publicly available.[2] As compiler we use g++ 4.8.3. The benchmark platform is a dual-socket server with two 8-core Intel Xeon E5-2680 at 2.7 GHz each and 256 GB RAM. We present a representative subset of our experiments,

[2]Information: http://parco.iti.kit.edu/software-en.shtml,
code: https://algohub.iti.kit.edu/parco/NetworKit/NetworKit-SDD.

in which we compare our KOSZ implementation to existing linear solvers as implemented by the libraries Eigen 3.2.2 [15] and Paralution 0.7.0 [23], both libraries with fast sparse matrix solvers.

We mainly use two graph classes for our tests: (i) Rectangular $k \times l$ grids given by $\mathbb{G}_{k,l} := \left([k] \times [l], \left\{\{(x_1, y_1), (x_2, y_2)\} \subseteq \binom{V}{2} : |x_1 - x_2| = 1 \vee |y_1 - y_2| = 1\right\}\right)$. Laplacian systems on grids are, for example, crucial for solving boundary value problems on rectangular domains; Note that $\mathbb{G}_{k,l}$ is very uniform, i. e. most of its nodes have degree 4. (ii) Barabási-Albert [5] random graphs with parameter k. These random graphs are parametrized with a so-called *attachment* k. Their construction models that the degree distribution in many natural graphs is not uniform at all.

For both classes of graphs, we consider both unweighted and weighted variants (uniform random weights in $[1, 8]$). We also did informal tests on 3D grids and graphs that were not generated synthetically. These graphs did not exhibit significantly different behavior than the two graph classes above and are therefore omitted from the presentation of the results.

Termination and performance counters. In the description of the solver so far we did not state our termination condition and Kelner et al. [19] only give a theoretical expected number of iterations to achieve a desired error in $\|\cdot\|_L$. We choose, as usual in iterative solvers, to terminate when the relative residual $\|Ax - b\|_2 / \|b\|_2$ is smaller than a given $\epsilon > 0$. Unfortunately, the KOSZ solver cannot keep track of the residual. To get it, we must first compute the dual potential vector x. Since this takes $\mathcal{O}\big(m \log(n)\big)$ time, we cannot update the residual every iteration. Therefore, to still get provably nearly-linear time, we heuristically choose to update it every m iterations. Informal experiments show that computing the residuals takes less than 3% of the global time and that only updating every m iterations does not prolong convergence more than 4% in all of our tests.

CPU performance characteristics such as the number of executed FLOPS (floating point operations), etc. are measured with the PAPI library [10]. Each of our benchmarking runs takes several seconds (billions of cycles), so we expect the counter values to be quite accurate. Moreover, our most basic choice to reduce the impact of possible measurement errors is to repeat the benchmark multiple times and average the values gathered. In our case, we repeated each measurement 10 times. This number is mainly motivated by time constraints. Since the resulting measurements are not skewed, we believe that the central limit theorem (an asymptotic theorem) is already applicable for these 10 runs. Given that the measured standard deviations are below 5%, the real counter values are within $-\operatorname{erf}(0.025) \cdot 5\%/\sqrt{10} \approx 3\%$ of the measured mean value with 95% confidence.

In addition, we take an optimistic approach with regards to cache usage and start each series of runs with a dry run that fills the caches.

4.2 Results

Spanning tree. Papp [26] tested various low-stretch spanning tree algorithms and found that in practice the provably good low-stretch algorithms do not yield better stretch than simply using Kruskal. We confirm and extend this observation by comparing our own implementation of Elkin et al.'s [14] low-stretch ST algorithm to Kruskal and Dijkstra in Figure 3. Except for the unweighted 100×100 grid, Elkin has worse stretch than the other algorithms and Kruskal yields a good ST. For Barabási-Albert graphs, Elkin is extremely bad (almost factor 20 worse). Interestingly, Kruskal outperforms the other algorithms even on the unweighted Barabási-Albert graphs, where it degenerates to choosing an arbitrary ST. Figure 3 also shows that our special ST yields significantly lower stretch for the unweighted 2D grid, but it does not help in the weighted case.

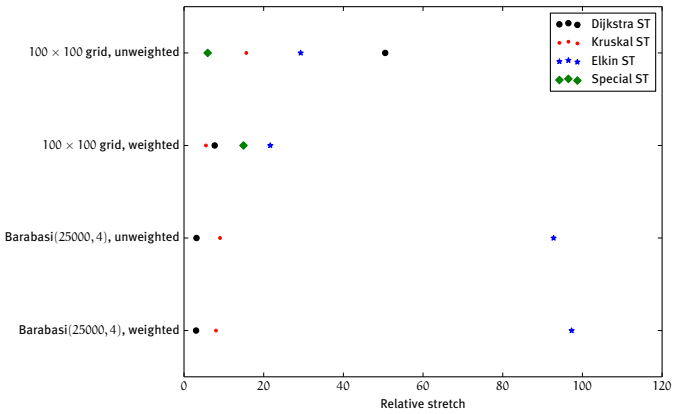
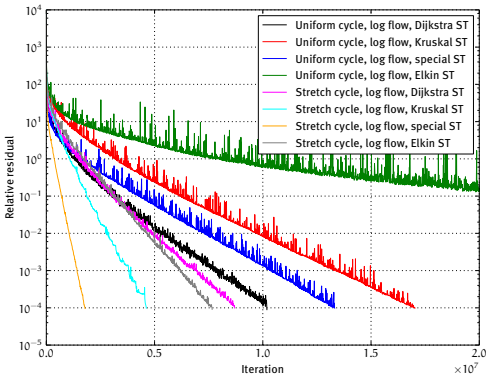


Figure 3: Average stretch $\mathrm{st}(T)/m$ with different ST algorithms.

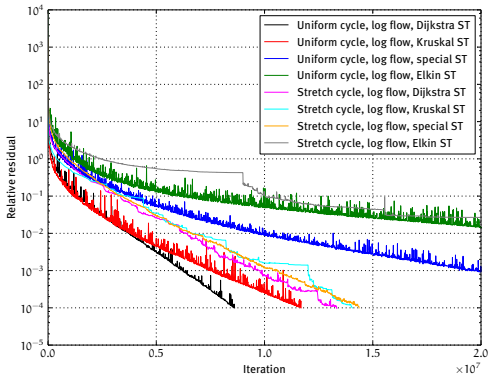
Convergence. In Figure 4 we plot the convergence of the residual for different graphs and different algorithm settings. We examine a 100×100 grid and a Barabási-Albert graph with $25{,}000$ nodes. In this experiment we determine the energy gap $\xi_r(f) - \xi_r(f_{\text{opt}})$ by fixing the optimal solution x and taking Lx as right hand side, i. e. $\xi_r(f_{\text{opt}}) = \zeta_r(x)$. As expected, the energy in all runs decreases monotonically. While the residuals can increase, they follow a global downward trend. Also note that the spikes of the residuals are smaller if the convergence is better and that the order (by convergence speed) of the residual curves and the energy curves is the same.

In all cases the solver converges exponentially, but the convergence speed crucially depends on the solver settings. If we select cycles by their stretch, the order of the convergence speeds is the same as the order of the stretches of the ST (cmp. Figure 3), except for the Dijkstra ST and the Kruskal ST on the weighted grid. In particular, for the Elkin ST on Barabási-Albert graphs, there is a significant gap to the other settings where the solver barely converges at all and the special ST wins. Thus, low-stretch STs are crucial for convergence. In informal experiments we also saw this behavior for 3D grids and non-synthetic graphs. In contrast, for the uniform cycle selection on the unweighted grid, the special ST is superior over the Kruskal ST, even though its stretch is smaller. This is caused by the fact that the basis cycles with the Kruskal ST are longer than the basis cycles with the special ST and fixing them helps more. Still, the other curves with uniform cycle selection follow the stretch.

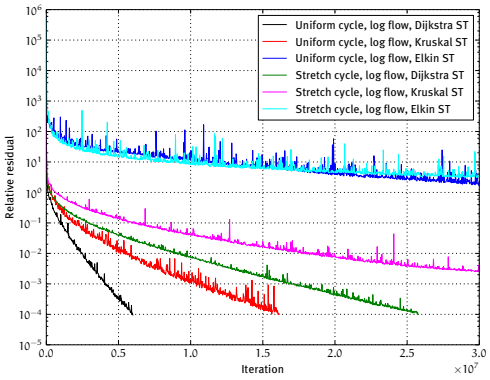
Using the results of all our experiments, we are not able to detect any correlation between the improvement made by a cycle repair and the stretch of the cycle. Therefore, we cannot fully explain the different speeds with uniform cycle selection and stretch cycle selection. For the grid the stretch cycle selection wins, while Barabási-Albert graphs favor uniform cycle selection. Another interesting observation is that most of the convergence speeds stay constant after an initial fast improvement at the start to about residual 1. That is, there is no significant change of behavior or periodicity. Even though we can hugely improve convergence by choosing the right settings, even the best convergence is still very slow, e.g. we need about 6 million iterations (≈ 3000 sparse matrix-vector multiplications (SpMVs) in time comparison) on a Barabási-Albert graph with $25{,}000$ nodes and $100{,}000$ edges in order to reach residual 10^{-4}. In contrast, conjugate gradient (CG) without preconditioning only needs 204 SpMVs for this graph (preconditioning is explained in the corresponding subsection below).



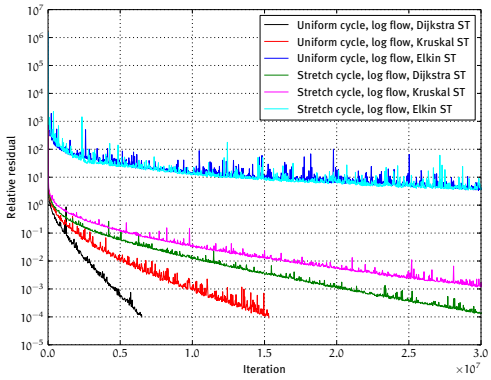
(a) 100×100 grid, unweighted



(b) 100×100 grid, weighted



(c) Barabási-Albert, $n = 25000$, unweighted

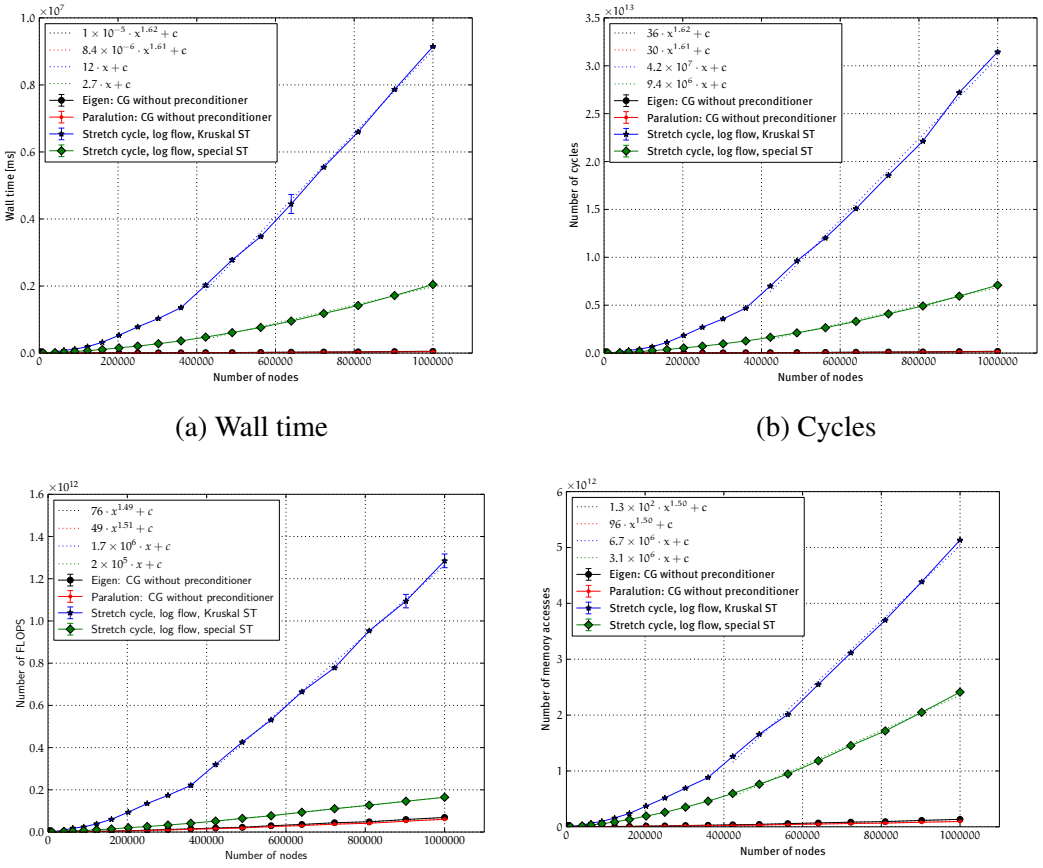


(d) Barabási–Albert, $n = 25000$, weighted

Figure 4: Convergence of the residual. Terminate when residual $\leq 10^{-4}$.

Asymptotics. Now that we know which settings of the algorithm yield the best performance for 2D grids and Barabási-Albert graphs, we proceed by looking at how the performance with these settings behaves asymptotically and how it compares to conjugate gradient (CG) without preconditioning, a simple and popular iterative solver (often used in its preconditioned form). Since KOSZ turns out to be not competitive, we do not need to compare it to more sophisticated algorithms.

In Figure 5 each occurrence of c stands for a new instance of a real constant. We expect the cost of the CG method to scale with $\mathcal{O}(n^{1.5})$ on 2D grids [12], while our KOSZ implementation should scale nearly-linearly. This expectation is confirmed in the plot: Using Levenberg-Marquardt [24] to approximate the curves for CG with a function of the form $ax^b + c$, we get $b \approx 1.5$ for FLOPS and memory accesses, while the (more technical) wall time and cycle count yield a slightly higher exponent $b \approx 1.6$. We also see that the curves for our KOSZ implementation are almost linear from about 650×650. Unfortunately, the hidden constant factor is so large that our algorithm cannot compete with CG even for a 1000×1000 grid. Note that the difference between the algorithms in FLOPS is significantly smaller than the difference in memory accesses and that the difference in running time is larger still. This suggests that the practical performance of our algorithm is particularly bounded by memory access patterns and not by floating point operations. This is noteworthy when we look at our special spanning tree for the 2D grid. We see that using the special



(a) Wall time

(b) Cycles

(c) FLOPS

(d) Memory accesses

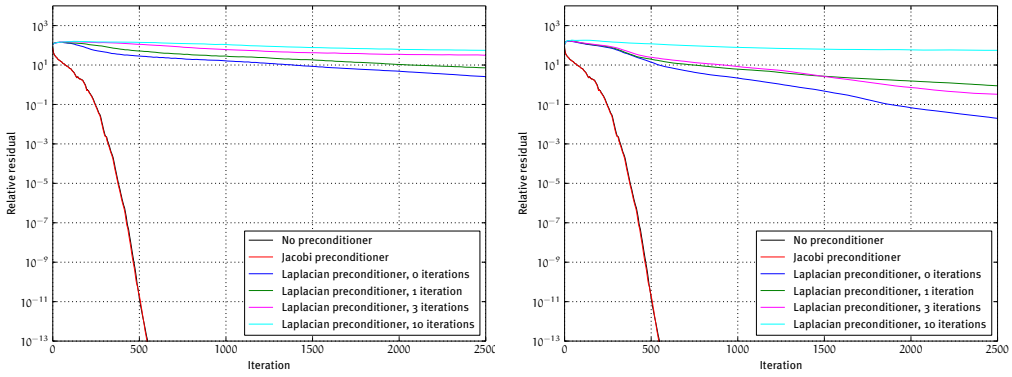
Figure 5: Asymptotic behavior for $2D$ grids. Termination when relative residual was $\leq 10^{-4}$. The error bars give the standard deviation.

ST always results in performance that is better by a constant factor. In particular, we save a lot of FLOPS (factor 10), while the savings in memory accesses (factor 2) are a lot smaller. Even though the FLOPS when using the special ST are within a factor of 2 of CG, we still have a wide chasm in running time. But note that later in this section we see that the micro-performance of the solver is actually very competitive with CG. Thus, the bad running time is mainly caused by memory accesses and the very slow convergence that we have already seen before.

The results for the Barabási-Albert graphs are basically the same (and hence not shown in detail): Even though the growth is approximately linear from about 400,000 nodes, there is still a large gap between KOSZ and CG since the constant factor is enormous. Also, the results for the number of FLOPS are again much better than the result for the other performance counters.

In conclusion, although we have nearly-linear growth, even for 1,000,000 graph nodes, the KOSZ algorithm is still not competitive with CG because of huge constant factors, in particular a large number of iterations and memory accesses.

Preconditioning. The convergence of most iterative linear solvers on a linear system $Ax = b$ depends on the *condition number* $\kappa(A)$ of A. The smaller the condition number is, the better the



(a) CG method, Kruskal ST

(b) CG method, special ST

(c) FGMRES method, Kruskal ST

(d) FGMRES method, special ST

Figure 6: Convergence of the residual when using the Laplacian solver as a preconditioner on an unweighted 100×100 grid.

solvers converge. A common way to improve the condition number is to find a matrix P such that $\kappa(P^{-1}A) < \kappa(A)$ and then solve the system $P^{-1}Ax = P^{-1}b$ instead of $Ax = b$ (*preconditioning*). Some linear solvers, such as Gauss-Seidel, are good preconditioners even though they are slow when used on their own. Thus we check whether this is also true for KOSZ.

In iterative methods we usually do not explicitly compute $P^{-1}A$ but apply P^{-1} and A separately to the current vector in each iteration. In our case we use a few KOSZ iterations as a preconditioner in each iteration instead of taking a fixed matrix P. Since the solver only works for SDD matrices, we need to use an iterative solver that only passes SDD matrices to the preconditioner. We choose the CG method and the FGMRES method on an unweighted 100×100 grid. The convergence of the residual with these solvers is plotted in Figure 6.

For the CG method we see that, unfortunately, the more iterations we use, the more slowly the methods converge. Since the cycle repairs depend crucially on the right hand side and the solver is probabilistic, using the Laplacian solver as preconditioner means that the preconditioner matrix is not fixed but changes from iteration to iteration. Axelsson and Vassilevski [4] show why this behavior leads to convergence problems and propose a CG method with variable-step preconditioning to cope with it. In practice the flexible GMRES method is often more resistant

to these convergence problems. Since the initial vector on the special ST is very good, we get good convergence in Figure 6 when using zero iterations of the solver in FGMRES, a behavior that is obviously not generalizable. For more iterations of the Laplacian solver, FGMRES still has convergence problems, but it is somewhat better than CG.

We conclude that KOSZ is not suitable as a preconditioner for common iterative methods. It would be an interesting extension to check if the solver works in a specialized variable-step method.

Smoothing. One way of combining the good qualities of two different solvers is *smoothing*. Smoothing means to dampen the high-frequency components of the error, which is usually done in combination with another solver that dampens the low-frequency error components. It is known that in CG and most other solvers, the low-frequency components of the error converge very quickly, while the high-frequency components converge slowly. Thus, we are interested in finding an algorithm that dampens the high-frequency components, a good smoother. This smoother does not necessarily need to reduce the error, it just needs to make its frequency distribution more favorable. Smoothers are particularly often applied at each level of multigrid or multilevel schemes [9] that turn a good smoother into a good solver by applying it at different levels of a matrix hierarchy.

To test whether the Laplacian solver is a good smoother, we start with a fixed x with $Lx = b$ and add white uniform noise in $[-1, 1]$ to each of its entries in order to get an initial vector x_0. Then we execute a few iterations of our Laplacian solver and check whether the high-frequency components of the error have been reduced. Unfortunately, we cannot directly start at the vector x_0 in the solver. Our solution is to use *Richardson iteration*. That is, we transform the residual $r = b - Lx_0$ back to the source space by computing $L^{-1}r$ with the Laplacian solver, get the error $e = x - x_0 = L^{-1}r$ and then the output solution $x_1 = x_0 + L^{-1}r$.

Figure 7 shows the error vectors of the solver for a 32×32 grid together with their transformations into the frequency domain for different numbers of iterations of our solver. We see that the solver may indeed be useful as a smoother since the energies for the large frequencies (on the periphery) decrease rapidly, while small frequencies (in the middle) in the error remain.

In the solver we start with a flow that is nonzero only on the ST. Therefore, the flow values on the ST are generally larger at the start than in later iterations, where the flow will be distributed among the other edges. Since we construct the output vector by taking potentials on the tree, after one iteration x_1 will, thus, have large entries compared to the entries of b. In subplot (c) of Figure 7 we see that the start vector of the solver has the same structure as the special ST and that its error is very large. For the 32×32 grid we, therefore, need about 10000 iterations (≈ 150 SpMVs in running time comparison) to get an error of x_1 similar to x_0 even though the frequency distribution is favorable. Note that the number of SpMVs the 10000 iterations correspond to depends on the graph size, e.g. for an 100×100 grid the 10000 iterations correspond to 20 SpMVs.

While testing the Laplacian solver in a multigrid scheme could be worthwhile, the bad initial vector creates robustness problems when applying the Richardson iteration multiple times with a fixed number of iterations of our solver. In informal tests multiple Richardson steps lead to ever increasing errors without improved frequency behavior unless our solver already yields an almost perfect vector in a single run.

Micro-performance and parallelism. The nearly-linear running time of the Laplacian solver was proved in the RAM machine model. To get good practical performance on modern out-of-

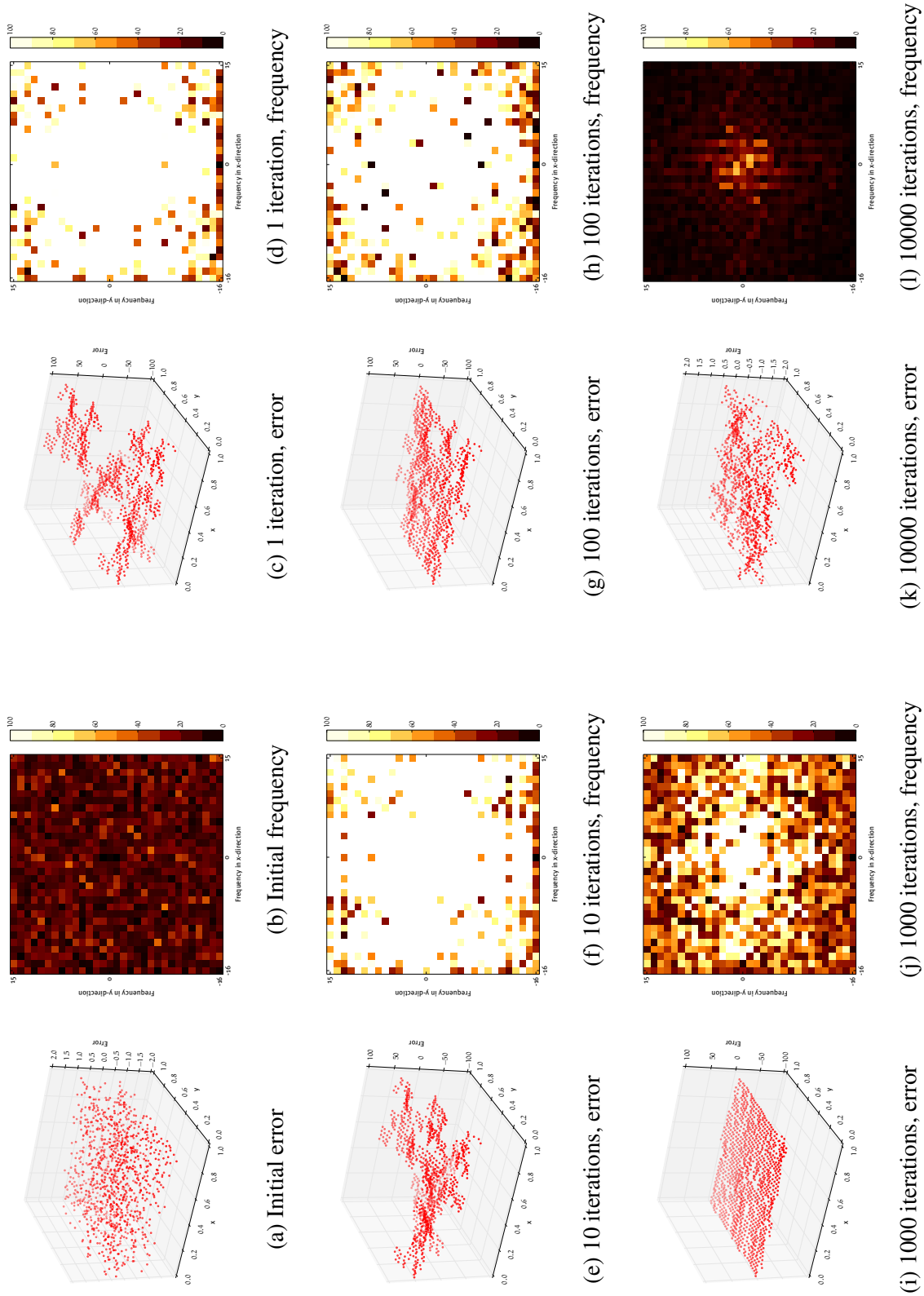


Figure 7: The Laplacian solver with the special ST as a smoother on a 32×32 grid. For each number of iterations of the solver we plot the current error and the absolute values of its transformation into the frequency domain. Note that (a) and (k) have a different scale.

order superscalar computers, one has to take their complex execution behavior into account, too.

One particular problem indicated by our experiments is that the number of cache misses increases in the LogFlow data structure when a bad spanning tree is used. Note that querying and updating the flow with this data structure corresponds to a dot product and an addition, respectively, of a dense vector and a sparse vector. The sparse vectors are stored as lists of pairs of indexes (into the dense vector) and values. Thus, the cache behavior depends on the distribution of the indexes, which is determined by the subtree decomposition of the spanning tree and the order of the subtrees.

We managed to consistently improve the time by about 6% by doing the decomposition in BFS order, so that the indexes are grouped together at the front of the vector. In contrast, the actual decomposition only depends on the spanning tree. Furthermore, we could save an additional 10% of time by using 256-bit AVX instructions to do four double precision operations at the same time in LogFlow, but this vectorized implementation still uses (vectorized) indirect accesses.

In our experiments we get about 5% cache misses by using the minimum weight ST on the 2D grid, compared with 1% when using CG. In contrast, the special ST yields competitive cache behavior. Not surprsingly, since the Barabási-Albert graph has a much more complex structure, its cache misses using the sparse matrix representation increase to 5%. In contrast, the cache misses improve for larger graphs with LogFlow since the diameter of the spanning tree is smaller than on grids and the decomposition, thus, groups most indexes at the start of the vector.

From the benchmarks we can infer that the micro-performance suffers from indirect accesses just as in the case of the usual sparse matrix representations. Furthermore, the micro-performance crucially depends on the quality of the spanning tree. For good spanning trees or more complex graphs, the micro-performance of the Laplacian solver is competitive with CG.

A discussion of its parallelization may seem unjustified given that the KOSZ solver was not designed for parallelism. Yet, a short treatment is appropriate since most performance improvements are achieved today by putting more cores on a chip, and other linear solvers are often executed in parallel. There are two basic ways of parallelizing the solver in a shared-memory setting, both of which do not scale well without changing its main loop or the flow data structure significantly:

First, we can parallelize each single query/update of the LogFlow data structure. Unfortunately, even for larger graphs the vectors involved are so sparse that parallelizing the operations never outweighed the cost of the barrier synchronization after each operation in our tests.

Second, we could also update multiple cycles at the same time. When we store each flow on an edge directly, each update consists of a query phase where we determine the amount of current to be added to the cycle and a cycle update phase. Between the phases the flow on the cycle needs to remain fixed. Thus, we need to lock whole cycles; atomic updates on single edges do not suffice. This would create significant synchronization overhead, but could still result in a viable parallelization if we manage to find many independent cycles. But we need to use the LogFlow data structure to get good provable and practical performance. This data structure works by decomposing a tree-path into two root-node paths in the decomposition tree. Since all of these paths intersect in the original tree, we cannot update them in parallel.

5 Conclusions

At the time of writing the conference version of this paper, we provided the first comprehensive experimental study of a Laplacian solver with provably nearly-linear running time. In the meantime,

our results regarding KOSZ have been recently confirmed and in some aspects extended [8].

Our study supports the theoretical result that the convergence of KOSZ crucially depends on the stretch of the chosen spanning tree, with low stretch generally resulting in faster convergence. This particularly suggests that it is crucial to build algorithms that yield spanning trees with lower stretch. Since we have confirmed and extended Papp's [26] observation that algorithms with provably low stretch do not yield good stretch in practice, improving the low-stretch ST algorithms is an important future research direction. Even though KOSZ proves to grow nearly linearly as predicted by theory, the constant seems to be too large to make it competitive, even compared to the CG method without preconditioner. Hence, regarding the paper title, we can say that the running time is nearly linear indeed and thus fast in the $\mathcal{O}$-notation, but the constant factors prevent usefulness in practice so far. While the negative results predominate, we hope to deliver insights that lead to further improvements, both in theory and practice. It seems promising to repair cycles other than just the basis cycles in each iteration, but this would necessitate significantly different data structures.

References

[1] I. Abraham, Y. Bartal, and O. Neiman. Nearly tight low stretch spanning trees. In *49th Annual Symposium on Foundations of Computer Science*, pages 781–790, 2008.

[2] I. Abraham and O. Neiman. Using petal-decompositions to build a low stretch spanning tree. In *44th ACM Symposium on Theory of Computing*, pages 395–406, 2012.

[3] N. Alon, R. M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24:78–100, 1995.

[4] O. Axelsson and P. Vassilevski. A black box generalized conjugate gradient solver with inner iterations and variable-step preconditioning. *SIAM J. on Matrix Analysis and Applications*, 12(4):625–644, 1991.

[5] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[6] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[7] E. Boman, B. Hendrickson, and S. Vavasis. Solving elliptic finite element systems in near-linear time with support preconditioners. *SIAM Journal on Numerical Analysis*, 46(6):3264–3284, 2008.

[8] E. G. Boman, K. Deweese, and J. R. Gilbert. Evaluating the potential of a laplacian linear solver. *CoRR*, abs/1505.00875, 2015.

[9] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial*. SIAM, 2000.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[11] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proc. 43rd ACM Symp. on Theory of Computing (STOC)*, pages 273–282. ACM, 2011.

[12] J. W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[13] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789–812, 1999.

[14] M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng. Lower-stretch spanning trees. In *Proc. of the 37th Annual ACM Symp. on Theory of Computing (STOC)*, pages 494–503. ACM, 2005.

[15] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[16] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, May 1984.

[17] D. Hoske, D. Lukarski, H. Meyerhenke, and M. Wegner. Is nearly-linear the same in theory and practice? A case study with a combinatorial laplacian solver. In E. Bampis, editor, *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 205–218. Springer, 2015.

[18] J. A. Kelner and A. Madry. Faster generation of random spanning trees. In *Proc.50th Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 13–21. IEEE Computer Society, 2009.

[19] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, pages 911–920, New York, NY, USA, 2013.

[20] I. Koutis. Simple parallel and distributed algorithms for spectral graph sparsification. In *Proc. 26th ACM Symp. on Parallelism in algorithms and architectures (SPAA)*, pages 61–66. ACM, 2014.

[21] I. Koutis, A. Levin, and R. Peng. Improved spectral sparsification and numerical algorithms for SDD matrices. In *Proc. Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 266–277, 2012.

[22] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7:48–50, Feb. 1956.

[23] D. Lukarski. Paralution - library for iterative sparse methods. 2015. http://www.paralution.com, last access: Nov 20, 2015.

[24] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[25] H. Meyerhenke and S. Schamberger. A parallel shape optimizing load balancer. In *Proc. 12th International Euro-Par Conference (Euro-Par 2006)*, pages 232–242. Springer, 2006.

[26] P. A. Papp. Low-Stretch Spanning Trees, 2014. Bachelor thesis, Eötvös Loránd University.

[27] R. Peng and D. A. Spielman. An efficient parallel solver for SDD linear systems. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14, pages 333–342, New York, NY, USA, 2014. ACM.

[28] J. Reif. Efficient approximate solution of sparse linear systems. *Computers & Mathematics with Applications*, 36(9):37 – 58, 1998.

[29] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983.

[30] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6):1913–1926, 2011.

[31] D. A. Spielman and S. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proc. 36th Annual ACM Symp. on Theory of Computing (STOC)*, pages 81–90, 2004.

[32] D. A. Spielman and J. Woo. A note on preconditioning by low-stretch spanning trees. *CoRR*, abs/0903.2816, 2009.

[33] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *arXiv:1403.3005*, 2014.

[34] P. M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Technical report, University of Illinois at Urbana-Champaign, Urbana, IL, 1990.