Parallel Graph Laplacian and Bundle Adjustment Solvers

by

Tristan Konolige

B.A., University of California Santa Barbara, 2015

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2020

This thesis entitled:
Parallel Graph Laplacian and Bundle Adjustment Solvers
written by Tristan Konolige
has been approved for the Department of Computer Science

Prof. Jed Brown

Prof. Lijun Chen

Prof. Christoffer Heckman

Prof. Thomas Manteuffel

Prof. Rebecca Morrison

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Konolige, Tristan (Ph.D., Computer Science)

Parallel Graph Laplacian and Bundle Adjustment Solvers

Thesis directed by Prof. Jed Brown

Multigrid is a powerful linear system solver for PDE systems. This thesis is concerned with expanding the domain of multigrid solvers to bundle adjustment, and expanding existing multigrid techniques for graph Laplacians into a distributed memory environment. Graph Laplacians are a mathematical model representing objects and relationships, and the interactions between them. When used as a model for social networks, graph structure is irregular, making solving the linear system difficult. Existing multigrid solvers for graph Laplacians are an effective solution, however, prior work is inherently serial. Parallel multigrid solvers have a rich history, but none are suited to solving graph Laplacian problems. In this work, the current state of the art is extended into a distributed memory environment by developing a new parallel-friendly aggregation algorithm and adjusting low-degree elimination for parallel datastructures. Parallel scalability on a large number of social network graphs shows good results.

Bundle adjustment is a nonlinear optimization technique used in Structure from Motion pipelines to remove error in camera and point observations. This thesis discusses the creation a new multigrid solver for bundle adjustment that improves performance on large, difficult problems. An analysis of what makes bundle adjustment problems difficult to existing solvers, and how multigrid addresses these problems is presented. Multigrid shows good scaling on large problems. Performance of various linear system solvers on bundle adjustment in distributed memory systems is also considered.

Dedication

For my parents.

Acknowledgements

The members of the Grandview gang: Tom, Steve, John, Aly, Ben and Ben, Chris, Jeff, Delyan, and Steffen, were very welcoming to me as a new student. Their explanation of the why and how of multigrid is something I would be unable to find anywhere else. And their hospitality at the Copper Mountain conferences is so very appreciated.

I'm thankful to my friends for maintaining my mental and social wellbeing. Eddie, Jesse, and Rob were amazing roommates and shared much of their lives with me. Mark and Trent were valuable outdoor companions. Jared provided endless conversation and distraction as well as useful input and advice. I am lucky to be friends with all of you.

I thank my advisor, Jed Brown, for his deep knowledge of most things mathematical and his understanding of my need for trips to the rivers and the mountains. The wilderness of Colorado provided many hours of beauty and room for thought.

Contents

Chapter

1 Introduction 1

1.1 Graph Laplacian . . . 2

1.1.1 Applications . . . 4

1.1.2 Social Network Graphs . . . 5

1.2 Bundle Adjustment . . . 6

1.2.1 Reprojection Error . . . 7

1.2.2 Levenberg-Marquardt . . . 7

1.3 Algebraic Multigrid . . . 8

1.3.1 Aggregation . . . 10

1.3.2 Strength of Connection . . . 12

1.3.3 AMG as a Preconditioner . . . 13

1.3.4 Measuring Performance . . . 14

1.4 Debugging Multigrid Performance . . . 14

1.4.1 Compatible Relaxation . . . 15

1.5 Distributed Memory Parallelism . . . 18

2 LigMG — A Parallel Graph Laplacian Solver 19

2.1 Related Work . . . 19

2.1.1 Direct Solvers . . . 19

2.1.2 Simpler Preconditioners 20

2.1.3 Theoretical Solvers 20

2.1.4 Practical Serial Solvers 20

2.2 Main Contribution 21

2.2.1 Issues With a Parallel Implementation of LAMG 21

2.2.2 2D Matrix Distribution 22

2.2.3 Random Vertex Ordering 25

2.2.4 Parallel Low-Degree Elimination 25

2.2.5 Parallel Aggregation 28

2.2.6 Smoothing 31

2.2.7 K-cycles 31

2.3 Numerical Results 32

2.3.1 Comparison to Serial 32

2.3.2 Strong Scaling 36

2.4 Conclusions 38

3 Synthetic Bundle Adjustment Problem Creation **39**

4 Multigrid for Bundle Adjustment **42**

4.1 Solving the Linear System 42

4.2 Related Work 45

4.3 Multigrid for Bundle Adjustment 45

4.3.1 Nullspace 45

4.3.2 Aggregation 46

4.3.3 Prolongation 48

4.3.4 Smoother 48

4.3.5 To Smooth or Not To Smooth 49

4.3.6 V-Cycles vs W-Cycles 50

4.3.7 Implicit Operator 50
4.4 Results 52
4.4.1 Solver Accuracy 54
4.4.2 Scaling 57
4.4.3 Eigenvalues 60
4.4.4 Parallelism 60
4.4.5 Robust Error Metrics 65
4.4.6 When to Use Multigrid 65
4.5 Conclusion & Future Work 66

5 Distributed Memory Bundle Adjustment **67**

6 Software Contributions **71**

7 Conclusion **72**

Bibliography **74**

Figures

Figure

1.1 Plot of magnitude of each component in the smallest compatible relaxation eigenvector of a bundle adjustment linear system. Only a few components in the vector have a large magnitude, indicating there are no long range effects present. 17

1.2 Plot of smallest eigenvector components on a per degree of freedom basis in a bundle adjustment linear system. Color of points indicates which aggregate they belong to. Black lines indicate direction of movement of the dof according to the smallest eigenvector. All of the large components are in a single aggregate, indicating that this aggregate should either be split or a stronger smoother should be used. There is no clear partition of the aggregate using the small eigenvector components, leading us to believe that a stronger smoother is needed. . 17

2.1 Boxplots of the performance of serial LAMG [45] on 110 graphs from the University of Florida Sparse Matrix collection [25]. Solver configuration (vertical axis) is a triple of smoother, iterate recombination (or not), and cycle index. Performance is measured in terms of work per digit of accuracy (see section 1.3.4). WDA accounts for work per iteration and number of iterations. Problems are solved to a relative tolerance of 10^{-8}. Each box represents the interquartile range of WDA for a given solver configuration. The line and dot inside the box indicates the median WDA. Horizontal lines on either side of the box indicate the range of WDA values. Dots outside the box indicate outliers. 23

2.2 The distribution of directed edges in the graph (left) and adjacency matrix (right). The top is a 1D vertex distribution, and the bottom is a 2D edge distribution. Each color corresponds to edges and matrix entries owned by processor. Note that no process in the 2D distribution has all the out-edges or in-edges for a given vertex. 23

2.3 Boxplots of solver performance in various configurations on a selection of 110 graphs from the University of Florida Sparse Matrix Collection [25]. Performance is measured in terms of work per digit of accuracy (see section 1.3.4). WDA accounts for work per iteration and number of iterations. The last number in the solver configuration indicates the cycle index. All solves to a relative tolerance of 10^{-8}. LAMG with Jacobi smoothing, no recombination, and cycle index 1 has many undisplayed outliers because they fall well above 100 WDA. . . . 33

2.4 Loglog plot of strong scaling of our solver with K-cycles and with V-cycles on the *hollywood* graph (1,139,905 vertices, 113,891,327 edges) on Edison. Numeric labels next to points indicate number of processes for a given solve. There are 21-23 multigrid levels so the coarsest level is visited $2^{10} - 2^{11}$ times with an index 2 cycle. The coarse level solves and redistribution become a parallel bottleneck. 35

2.5 Semilog-x plot of efficiency $\left(\frac{\mathrm{nnz}(L)}{\text{TDA}\cdot\text{number of processes}}\right)$ vs. solve time for a variety of large social network graphs on Cori. Solves are to a relative tolerance of 10^{-8}. Numeric labels next to points indicate number of processes for a given solve. *hollywood* took 15 iterations on 196 processors versus 13 iterations on all other processor sizes, leading to loss of efficiency and negligible speedup. Some solves on the same problem perform more iterations (and solve to a slightly higher tolerance) than others causing variation in efficiency. 37

2.6 Semilog-x plot of normalized efficiency $\left(\frac{\mathrm{nnz}(L)}{\text{time per work unit}\cdot\text{number of processes}}\right)$ vs time per digit of accuracy for a variety of large social network graphs on Cori. Numeric labels next to points indicate number of processors used for a given solve. 37

3.1 Left: 3D model of Zwolle (Netherlands). Right: generated synthetic bundle adjustment dataset. 39

4.1 Nonlinear objective function values vs cumulative solve times for multigrid with V-cycles and W-cycles on a set of increasingly larger synthetic problems. Multigrid performs about the same with V-cycles and W-cycles, indicating that coarse grid solves are accurate enough with V-cycles. 51

4.2 Preconditioner solve time versus multigrid solve time for a set of synthetic problems with varying number of cameras, visibility structure, and noise. Solve time is measured as total time spent in the linear solver (setup and solve) for all nonlinear iterations to a certain problem dependent tolerance. Points above the diagonal (black line) indicate the problem was solved quicker with multigrid than the given preconditioner, points below indicate that multigrid was slower. Vertical columns of plots have use the same loss function. Horizontal plot rows have the same linear solve tolerance τ. For the majority of cases, multigrid performs better than all the other solvers. 53

4.3 Objective function value vs nonlinear iteration number for a variety of synthetic problems with varying problem structure. Our multigrid solver tends to reach that value in fewer iterations than the other solvers because it is more accurate for a given solve tolerance. For solves where a high accuracy is required, or where Jacobian calculation is expensive, our solver is a good choice. 55

4.4 Objective function value vs cumulative time for a variety of synthetic problems with varying problem structure. Although multigrid is slow initially, its handling of long range error means it converges quickly for longer than point block Jacobi and visibility based preconditioning. Note that visibility based preconditioning is very slow in the first couple of iterations. We believe this is a scalability bug in its setup phase. 55

4.5 Residual norm vs error norm on a synthetic problem with long range noise. Note that for a given error, multigrid has a higher residual norm than point block Jacobi. 56

4.6 Linear solver scaling experiment on a series of increasingly larger grids with long range noise only. Grid size is on the order of $\sqrt{\text{number of cameras}} \times \sqrt{\text{number of cameras}}$. The y-axis is a measure of linear solver solve time (not including linear solver setup) per camera. A horizontal trend indicate that a solver is scaling linearly with the number of cameras. Slopes greater than zero indicates the solver is scaling superlinearly. We see the expected behavior that Multigrid scales close to linearly while visibility and point block Jacobi scale superlinearly. Smoothed aggregation multigrid has the best scaling, but its setup phase is prohibitively expensive. 58

4.7 Linear solver scaling experiment on a series of increasingly larger grids with long range noise only. The setup is the same as figure 4.6, except this plot contains the setup and solve times of the linear solver. Our multigrid preconditioner with smoothed aggregation performs worse than point block Jacobi and our multigrid preconditioner without smoothing due to the high setup cost of prolongations smoothing. The visibility preconditioner appears to be scaling as $O(n^3)$. Comparing to the plot without setup time, we see that this poor scaling appears entirely in the setup phase. 58

4.8 Linear solver scaling experiment on a series of increasingly larger grids with long range noise only. The setup is the same as figure 4.6, except this plot measures the iteration count of the solve vs the number of cameras. We would expect the smoothed aggregation multigrid to have zero slope as it should be an $O(n)$ solver. The noise introduced into the problem could contribute to the superlinear behavior we are seeing. 59

4.9 Nonlinear cost vs nonlinear iteration number on a 40 by 40 synthetic city grid. The linear solve tolerance is $\tau = 0.01$ with a Huber loss function to expose long range error. The cost for solves with point block Jacobi lags behind solves with multigrid or visibility preconditioners because point block Jacobi is a less accurate solver. 61

4.10 Eigenvector plots for a synthetic problem. In clockwise order from top left: 1. Smallest eigenvector on the first nonlinear solve iteration 2. Second smallest eigenvector on the first nonlinear iteration 3. Smallest eigenvector on the 10th nonlinear iteration 4. First and second largest eigenvectors on the first nonlinear iteration. The smallest eigenvectors remain the same between the first and 10th nonlinear solve, but their respect eigenvalues are different by a factor three orders of magnitude. 62

4.11 Condition number and linear solver iteration count vs problem diameter on a series of increasingly larger synthetic problems. Although the linear problems becoming increasing difficult to solve with larger diameter, the condition number does not reflect this. 63

4.12 Plot of the smallest eigenvector at the nonlinear solution of a synthetic bundle adjustment problem. Largest movements are in the direction that the cameras are facing. 64

5.1 Loglog plot of strong scaling of Levenberg-Marquardt on a set of synthetic problems. The y-axis measures efficiency as $\frac{\#\text{ of cameras}}{\text{time}\times\#\text{ of nodes}}$. Horizontal lines on this plot indicate perfect scaling. Numeric labels indicate number of processes used. 69

5.2 Loglog plot of weak scaling of Levenberg-Marquardt on a set of synthetic problems. The y-axis measures efficiency as $\frac{\#\text{ of cameras}}{\text{time}\times\#\text{ of nodes}}$. Horizontal lines on this plot indicate perfect scaling. 70

Chapter 1

Introduction

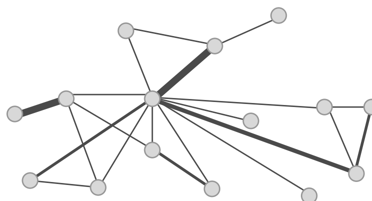
This thesis covers two main areas: graph Laplacians and bundle adjustment. Graph Laplacians are linear systems used in analysis and embedding of networks. Bundle adjustment is a nonlinear optimization problem used for creating 3D maps. Although these problems are very different, they share a common feature: both require the solution of linear systems of very large size. They also both have some sort of irregular structure that can be exploited. The solver of choice for large linear systems is multigrid, a multilevel approach. The typical area for multigrid use is partial differential equations (PDEs), used in various kinds of simulation and modeling. These systems are regular in their connectivity: degrees of freedom are connected to a low number of neighbors that are close in the physical domain. Both graph Laplacians and bundle adjustment are removed in some way from the typical area of multigrid usage. Graph Laplacians contain irregular structure not seen in most PDEs and bundle adjustment is a block system that combines similar irregularity with a richer near-nullspace. In both these cases, the different structure poses challenges to existing multigrid solvers.

Although multigrid scales linearly with problem size, some problems exceed the limits of computation that can be done on a single computer. Stepping to distributed memory parallelism increases the size of problems that can be solved in a reasonable amount of time. Although multigrid is often used in a distributed memory environment, graph Laplacians and bundle adjustment have structure that makes the usual approaches infeasible. This thesis will discuss an approach to distributed memory parallelism for multigrid solver for graph Laplacians as well a serial multigrid solver for bundle adjustment. This thesis

will also discuss distributed memory parallel solvers for bundle adjustment.

1.1 Graph Laplacian

Graphs arise in many areas to describe relationships between objects. In a graph, there are two types of objects: vertices and edges that connect them. Here is a visual representation of a graph:



with the grey circles representing vertices and the lines between them representing edges. These could be, for example, people in a social network (vertices) and friendships between them (edges), computers and the network connections between them, or cities and the highways connecting them. In all these cases, the structure of the graph depends on how these connections are formed. In a social network graph, a famous person would be connected to many more people than a regular person leading to a hub-like structure where one vertex is connected to many others. In the case of a road network, there could be long chains corresponding to a series of connected cities. No matter the case, the underlying geometry or characteristic of the problem is reflected in the graph structure. The structure of a graph can be characterized in many ways. The diameter of a graph measures the longest shortest path in the graph. This measure reflects how long it takes information to propagate in the graph. The degree of a vertex is the number of edges connected to it. This is a measure of local connectedness of a vertex.

In this thesis we will consider only weighted, undirected graphs. A weighted graph has a weight associated with each edge. Heavier edges may represent stronger connections. An undirected graph is one in which the edges are bidirectional. For example, an undirected road network would mean that one can travel both ways on every street. A weighted graph $G = (V, E, w)$ (where V are vertices, E are edges and

w are edge weights) can be expressed as an adjacency matrix A:

$$A_{ij} = \begin{cases} w_{ij}, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

The Laplacian matrix L can then be expressed as:

$$L = D - A, \tag{1.1}$$

$$D_{ij} = \begin{cases} \sum_u A_{uj}, & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{1.2}$$

As the graph is undirected, $(i,j) \in E \iff (j,i) \in E$ and $w_{ij} = w_{ji}$. We also limit ourselves to positively weighted ($w \geq 0$) graphs.

Graph Laplacians of this type have a couple of properties:

- Column and row sums are zero.
- Off diagonal entries are negative.
- Diagonal entries are positive.
- L is symmetric positive semi-definite.

In this thesis, we assume that G is connected. A connected graph is one where each vertex can reach every other vertex by traversing edges. For example, in a connected road network, every city can be reached from every other city by driving on road. A graph with multiple connected components can be considered as a set of independent graphs and each can be solved separately. Determining connected components is a preprocessing step, so we ignore it and assume all graphs we will be dealing with are connected.

For a connected graph, the corresponding Laplacian matrix has a null space spanned by the constant vector. The dimension one nullspace is easier to keep track of and orthogonalize against. Given the eigensystem $Lu_i = \lambda_i u_i$, the eigenvalues $0 = \lambda_0 \leq \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_{n-1}$ are non-negative and real. The

multiplicity of zero eigenvalues is equal to the number of connected components in the graph. Because we are only considering connected graphs the multiplicity of zero eigenvalues is one. The eigenvector u_1 associated with the second smallest eigenvalue approximates the sparsest cut—a partitioning of the graph into two halves that minimizes the number of edges cut over the number of vertices in the smaller half. The sign of u_1 determines which side of the cut each vertex belongs to [62].

1.1.1 Applications

Some applications of the graph Laplacian include:

- **Graph partitioning** As mentioned above, the eigenvector associated with the second second smallest eigenvalue of the graph Laplacian approximates the sparsest cut in the graph (Assuming a single connected component).

- **Graph drawing** Like graph partitioning, spectral graph drawing relies on finding eigenvalues and eigenvectors of the Laplacian to embed the graph in a two dimensional space for visualization. Each vertex i is placed at $(u_1(i), u_2(i))$. Other graph drawing techniques, such as the Maxent-Stress model, rely on solving the graph Laplacian [28].

- **PDEs on unstructured meshes** Some discretizations of partial differential equations on unstructured meshes result in matrices of the same from as a graph Laplacian [63]. If the mesh has large variation in vertex degree, solution techniques for arbitrary graph Laplacians might outperform those designed for fixed degree or more structured meshes.

- **Electrical flow** A circuit of resistors with current inputs and sinks between them can be modeled as a graph Laplacian. Each vertex in the graph is a current input or sink, and edges correspond to resistors with resistance equal to reciprocal resistance. If r is a vector where r_i is the current input or draw, then solving $p = L^{-1}r$ gives the potential p_i at vertex i [63].

See Spielman's article on applications of the graph Laplacian for more details [63].

All these applications (either directly or indirectly) require applying the action of the inverse of the graph Laplacian (L^{-1}). We refer to computing x in $Lx = b$ as *solving* the graph Laplacian—the equivalent of applying the inverse. We refer to a method of solving $Lx = b$ as a solver. Often, a highly accurate solve is not required. In these cases, a cheap iterative method is applied to compute an approximation of x. For example, to compute the small eigenvalues and associated eigenvectors for graph partitioning, an eigensolver iterates on approximations of inverted matrix, L^{-1}, using an iterative method. In this use case, the majority of time is spent in the linear solve (i.e., computing an approximation to $L^{-1}x$).

1.1.2 Social Network Graphs

There are many different kinds of graphs: road networks, computer networks, social networks, power grids, document relations, etc. We focus our performance evaluation on social network graphs because they often provide the most difficulty for existing multigrid solvers. Structured graphs, such as those associated with PDEs discretized on a grid, provide more opportunities for a solver to cut corners. Social networks have a couple of properties that make them more difficult to solve than other graphs:

- Social network graphs are sparse: the number of edges is roughly a constant factor of the number of vertices. This property allows us to use a sparse solver on the graph Laplacian to solve it in hopefully $O(|V|)$ time (theoretically possible for some sparse linear systems). If the graph was not sparse, the best we could hope is a solve time of $O(|V|^3)$ for a direct solver.

- Edges in these networks follow a power-law degree distribution (these graphs are called scale-free). A small number of vertices have a large number of neighbors, whereas the rest have a relatively small degree. This presents a challenge when determining how to distribute work in parallel.

- These graphs are expander-like. An expander graph is one in which a random relatively small subset of the vertices has a large number of neighbors. This causes difficulty in linear solvers that rely on the neighbors of neighbors of a vertex as this neighborhood of neighborhood encompasses almost the entire graph.

1.2 Bundle Adjustment

Large scale mapping applications such as Google Street View [36] and maps for self driving cars rely on triangulation of features in images to accurately create 3D maps. This construction of 3D maps from image data is referred to as *Structure from Motion* (SfM). An SfM pipeline works as follows:

(1) A sequence of images is captured from moving vehicles or hand-held cameras.

(2) A sparse set of features are extracted from each image.

(3) Features are matched across pairs of images. Not all pairs of images contain matches.

(4) Feature matches along with (optional) GPS data are used to triangulate the approximate locations the images were taken from (camera poses) and locations of features in the world (3D points).

(5) Approximate locations along with feature matches are fed into a global optimization pass that jointly refines camera poses and 3D points to high accuracy. This processes is called *bundle adjustment*.

Feature matching (step 3) and bundle adjustment (step 5) are usually the computationally expensive parts of Structure from Motion. Feature matching is a hard problem as the naive approach matches every image with every other image. However, it is trivially parallelizable, so it is not (theoretically) the limiting factor on large datasets. For this reason, and the fact that we are interested in exploiting hierarchical structure, we focus on bundle adjustment.

Step 4 above results in a noisy approximation of camera poses and world points. To refine these approximations in a globally consistent way, a nonlinear optimization step called bundle adjustment is used. We use a nonlinear least-squares formulation of bundle adjustment where we minimize

$$\min_x \frac{1}{2} \sum_i \|y_i - f(x_i)\|^2_,$$

where y are some observation coordinates in images, f is the model function, and x are the parameters. For bundle adjustment the model function is projection and the parameters are camera poses, camera intrinsics, point locations, and observations. Camera poses consist of a rotation and a translation in 3D space.

Camera intrinsics model how points in the world are projected into the camera image. Intrinsics include a focal length, and two radial distortions. Point locations are translations in 3D space. The observation coordinates, y, are the 2D coordinates (u, v) of the feature in the camera image frame. *Bundle Adjustment – a Modern Synthesis* [70] provides a good overview of bundle adjustment.

1.2.1 Reprojection Error

We use projection error as our model function. This is a common choice that seeks to minimize the difference between each point projected into a camera and the actual location the point was observed at. Given a point p; a camera composed of a 3×3 rotation matrix R, a 3×1 translation vector t and a 3×3 intrinsic matrix K; and an observation at (u, v), the reprojection error is defined as:

$$x = \text{proj}\left(K \begin{bmatrix} R & t \end{bmatrix} \begin{bmatrix} p \\ 1 \end{bmatrix}\right), \tag{1.3}$$

$$h([z_1\ z_2\ z_3]^T) = \frac{[z_1\ z_2]^T}{z_3}, \tag{1.4}$$

$$\text{proj}(y) = \left(1 + ||y||^2 * \left(d_1 + d_2 * ||y||^2\right)\right) h(y), \tag{1.5}$$

$$\text{reprojection error} = (u - x_0)^2 + (v - x_1)^2, \tag{1.6}$$

where proj projects the point from homogeneous coordinates to inhomogeneous coordinates (including radial distortion). To avoid overparameterization of the rotation and intrinsic matrices, we represent the rotation as a 3×1 Rodrigues vector, the intrinsic matrix with a focal length, and the distortion with two parameters (d_1, d_2). The reprojection error contains two residuals per observation: one for u and one for v.

1.2.2 Levenberg-Marquardt

A usual choice of solver for the nonlinear least-squares bundle adjustment problem is the Levenberg-Marquardt algorithm [42, 48]. This is a quasi-Newton method that repeatedly solves $J^T J + D = -Jr$ where J is the Jacobian of the model function, D is a diagonal damping matrix, and r is the residuals. Levenberg-Marquardt can be considered as a combination of a Gauss-Newton with gradient descent.

Algorithm 1 Levenberg-Marquardt Optimization

```
1:  function LM(initial solution x, objective function f, Jacobian function J)
2:      λ ← λ_0
3:      while not converged do
4:          D ← λI
5:          solve (J(x)^T J(x) + D)s = −J(x)^T x for s        ▷ Linear solve
6:          if f(x + s) < f(x) then                              ▷ Successful step
7:              x ← x + s
8:              λ ← λ/2
9:          else                                                 ▷ Failed step
10:             λ ← 2λ
11:         end if
12:     end while
13: end function
```

This is a simple version of Levenberg-Marquardt. There are many enhancements that can be made; for example λ can be updated depending on how many successful or unsuccessful steps have occurred [74]. Or it can be updated based on the size of the step and value of the objective function [69]. The damping matrix, D, does not necessarily need to be a scaling of the identity matrix. One can use a scaling of the diagonal of $J(x)^T J(x)$ instead [42]. Sometimes, a line search can be used to improve results [22].

1.3 Algebraic Multigrid

Algebraic multigrid (AMG) is a family of techniques for solving linear systems of the form $Ax = b$. Specifically, algebraic multigrid constructs a hierarchy of approximations to A using only the values in the matrix A. These approximations, $\{A = A_0, A_1, A_2, A_3, ...\}$, are successively coarser: size$(A_l) >$ size(A_{l+1}). For each level l in the hierarchy, we have a restriction operator R_l that transfers a residual from level l to level $l + 1$ and a prolongation P_l that transfers a solution from level $l + 1$ to level l. The coarse level matrix is usually constructed via a Galerkin product: $A_{l+1} = R_l A_l P_l$.

Algorithm 2 Multigrid cycle with cycle index γ

```
1:  function MGCYCLE(level l, initial guess x, rhs b)
2:      if l is the coarsest level then
3:          x ← Direct solve on A_l x = b
4:          return x
5:      else
6:          x ← smooth(x, b)                          ▷ Pre-smoothing
7:          r ← b − A_l x                             ▷ Residual
8:          r_c ← R_l r                               ▷ Restriction
9:          x_c ← 0
10:         for i ∈ [0, γ) do
11:             x_c ← MGCYCLE(l + 1, x_c, r_c)        ▷ Coarse level solve
12:         end for
13:         x ← x + P_l x_c                           ▷ Prolongation
14:         x ← smooth(x, b)                          ▷ Post-smoothing
15:         return x
16:     end if
17: end function
```

Algorithm 2 depicts a multigrid cycle with cycle index γ ($\gamma = 1$ is called a V-cycle, and $\gamma = 2$ is a W-cycle). Repeated application of this algorithm $x^{k+1} \leftarrow \text{MGCYCLE}(0, x^k, b)$ often converges to an approximate solution $Ax^* \approxeq b$, reaching a given tolerance in a number of iterations that is bounded independent of problem size. Fast convergence depends on sufficiently accurate restriction and prolongation operators complemented by pre-smoothing and post-smoothing that provide local relaxation. Typical smoother choices are Gauss-Seidel, Jacobi, and Chebyshev iteration. Note that pre- and post-smoothing may use different smoothers or different numbers of smoothing iterations. Given a restriction and prolongation, we identify “low frequencies” as those functions that can be accurately transferred to a coarse space and back. A smoother need only be stable on such functions, but must reduce the error uniformly for all “high frequencies”—those which cannot be accurately transferred.

There are two main ways of constructing $\{A_1, A_2, A_3, ...\}$: classical AMG and aggregation-based AMG. Classical (or Ruge-Stüben) AMG constructs R and P using a coarse-fine splitting: the coarse grid is a subset of the degrees of freedom (or “points”) of the fine grid [59]. P keeps values at coarse points and extends them via a partition of unity to neighboring fine points. Then, $R = P^T$. In aggregation-based AMG, degrees of freedom are clustered into aggregates. An aggregate on the fine level becomes a point on the coarse level. R and P usually take some weighted average of points in each aggregate to the coarse level. Aggregation-based multigrid methods typically smooth R and P for better performance [72].

In this thesis we use exclusively aggregation-based AMG. For the irregular domains we cover, finding representative degrees of freedom in classical AMG can pose a challenge. Furthermore, in situations where slow to converge modes are known, classical AMG does not provide a method for improving the coarse grid approximation. Aggregation-based AMG provides a clear method for using slow to converge modes in addition to having some theory on how to build aggregates on irregular problems.

1.3.1 Aggregation

There are many choices of how to choose aggregates for aggregation based multigrid methods. All aggregation routines attempt to aggregate degrees of freedom such that there is maximal connectivity inside the aggregate and minimal connectivity outside the aggregate—essentially a graph clustering problem. Routines must trade off between aggregate size and convergence speed. Large aggregates make coarse levels smaller, but are slower to converge. On the other hand, small aggregates converge quickly, but coarse levels are larger.

Degrees of freedom within an aggregate should be tightly coupled so that their coarse level degree of freedom is accurately representative. Most aggregation routines use a strength-of-connection metric to determine what constitutes a tightly coupled set of dofs (see section 1.3.2). This metric is expressed as a weighted graphs where the weight of an edge determines the strength of the connection. The connectivity of this graph is a subset of the connectivity of the dofs in the linear problem. Connections not in the linear problem are not considered as they are at most indirect. Aggregation routines for more regular meshes usually expect that all connections in the strength-of-connection matrix are "strong". Poor connections are then dropped before aggregation, either using a constant threshold or some row-based threshold [53, 56, 59, 72]. This thresholding is not robust—different thresholds can have significant impact on performance of the solver. The correct choice of threshold depends on mesh shape and the type of problem being solved. Aggregation routines intended for less regular problems usually do not require thresholding. These routines are intended to work with a variety of strength-of-connection weights and attempt to choose the heaviest edges before the lighter ones.

There are a wide range of aggregation routines; we focus on the following important ones standard [72],

pairwise [53, 54], and LAMG's [46] aggregation. A typical aggregation routine used for more regular meshes is what we will call "standard" aggregation. Standard aggregation first selects a dof with no aggregated neighbors. This dof becomes a root, and all neighboring dofs become part of its aggregate. The process repeats until no more dofs can be converted to roots. Remaining dofs are aggregated to a neighboring aggregate. It is clear that this algorithm needs a filtered strength-of-connection matrix. Otherwise, poorly connect dofs could become roots.

Pairwise aggregation, also known as heavy edge matching, is a routine that does not use a thresholded strength-of-connection matrix [53]. It chooses an unaggregated dof and aggregates it with the closest (by strength-of-connection) unaggregated neighbor. This is repeated until no unaggregated dofs exist. Stopping here can yield good aggregates, but this coarsening is not aggressive enough. The coarse grids are often too large to make up for the good convergence factors. To make larger aggregates, the process can be repeated by joining pairs of aggregates [54]. Although this routine does not require thresholding, it does not work well on problems with irregular connectivity. For example, star-like connectivity in the strength-of-connection graph results in only one of many dofs getting aggregated.

An example of an aggregation routine that works on irregular connectivity is the routine from LAMG [46]. This routine picks dofs with strong connections as "seeds" which form the center of an aggregate. Dofs with strong connections to a seed are aggregated with it. This is repeated in rounds with a decreasing threshold for what constitutes a strong connection until all dofs are aggregated. This avoids a need for an explicit threshold for the strength-of-connection matrix and also avoids problems pairwise-aggregation faces on irregular connectivity.

These three examples are in no way all the existing aggregation algorithms. Although "standard" aggregation is used in many places, it has many variants and no definitive version. On more unstructured problems, the list of aggregation routines is vast. The heuristics used for choosing a routine depend on the structure the solver expects and the type of problem being solved.

Aggregation routines face another constraint in that they need to be used in parallel environments. Because multigrid solve time scales linearly with problem size, it is a good choice for large problems. To decrease solve times, large problems are solved in parallel on distributed memory machines. Most of multi-

grid is readily amenable to a distributed memory environment. Prolongation construction, coarse grid construction, smoothing (excluding Gauss-Seidel smoothing [3, 9]), and residual evaluation are all straight forward linear algebra, so using a distributed linear algebra library gives parallelism without any extra work. The one part of multigrid that is not trivially made parallel is aggregation. Aggregates can be formed on a per process basis, but ignoring connections that span processes results in poor performance (especially in the limiting case where there is only one dof per process) [71]. One solution is to use a parallel maximum independent set algorithm to choose aggregates from the strength-of-connection graph [23, 31]. Parallel maximum independent sets can form aggregates regardless of the distribution of dofs between processors. However, parallel maximum independent sets performs poorly in places were connectivity is irregular because aggregate sizes are too large.

1.3.2 Strength of Connection

The strength-of-connection metric determines how "close" any two dofs are. The difficulty is deciding what constitutes "close." Two factors make this more complicated: 1. the optimal closeness metric may depend on non-local information and 2. closeness affects the structure of the coarse levels and hence affects closeness on the coarse levels.

The simplest strength-of-connection metric to use is entries from the linear system itself. If the connectivity is regular (fixed degree) and the problem is not anisotropic, this can be a good metric. However, it breaks down when the problem is not tightly coupled in a uniform, grid-aligned direction. Many more sophisticated strength-of-connection metrics have been proposed. Some are problem-dependent and use supplemental information, while others are black-box and only use entries in the linear system itself. Metrics that are able to exploit some extra information from the problem are often more accurate than those that do not. Like in many other parts of multigrid, using extra information leads to improved performance over using a black-box approach, but is not out of the box compatible with new applications.

One way to construct robust strength-of-connection metrics in a black-box approach is to use test vectors. These are vectors filled initially with random data (usually in the range $[-1, 1]$) and smoothed on $Ax = 0$ (using the smoother that will be used in the multigrid method). Comparing entries in these vectors gives

an idea for how quickly information is dispersed on the local scale [56]. If two entries have close to the same value, then it is likely that they are closely linked. In the smoothing sense these entries are effectively solved using a couple applications of the smoother, so they should be grouped together on the coarse level. On the other hand, if these two entries have dissimilar values, it is likely that they are not closely linked. A smoother does not solve them, so keeping them separate on the coarse level means they can be solved there. A number of vectors are used in order to make the metric more robust in the face of randomness. The more vectors we use, the closer we get to the true mean value, but each vector increases the cost of the setup phase. Normally, a small (less than 10) number of vectors are used [46]. The choice of how many times to smooth the test vectors can be important. Smoothing the vectors too much makes the function locally uniform so the distance measures capture no information (they become zero) [56]. However, smoothing only once may not be enough as the test vectors still appear to be random.

Algebraic distance is a good example of a strength-of-connection metric that uses test vectors:

$$\frac{1}{\text{algebraic distance}_{uv}} = \left(\sum_{k=1}^{K} \left| X_u^k - X_v^k \right|^p \right)^{\frac{1}{p}}.$$

X is the $n \times K$ matrix of K test vectors mentioned above [58, 61]. Algebraic distance can have issues with complicated connectivity. Livne and Brandt show that in the case of two connected hubs (high degree dofs), algebraic distance incorrectly marks the hubs as close [46]. To fix these problems, they propose affinity:

$$\text{affinity}_{uv} = \frac{|(X_u, X_v)|^2}{(X_u, X_u)^2 (X_v, X_v)^2}, \tag{1.7}$$

$$(X, Y) = \sum_{k=1}^{K} X^k Y^k, \tag{1.8}$$

which scales the metric based on the relative magnitudes of the dofs [46].

1.3.3 AMG as a Preconditioner

AMG can be used as a stand alone solver, but it often provides faster and more robust convergence when used as a preconditioner for a Krylov method [66]. In order to solve $Ax = b$, the Krylov method (usually conjugate gradients [32] or GMRES [60]) is applied to $MAx = Mb$, where M is a single AMG cycle, such

as a V-cycle or W-cycle. This choice is called left preconditioning because M is applied on the left side of the operator. The convergence of the Krylov method will depend on the spectrum of MA, converging in a number of iterations bounded by square root of the condition number. The goal of M, then is to reduce the condition number of A (assuming A is symmetric).

1.3.4 Measuring Performance

In order to evaluate the performance of a solver relative to another solver, a performance metric is needed. Runtime is often used, but it is dependent on the implementation of the algorithms (for example, a MATLAB implementation would probably be slower than a C++ implementation). Instead, we will use work per digit of accuracy (WDA) to measure performance:

$$\text{WDA} = \frac{-\text{work}}{\log_{10} \Delta r}, \tag{1.9}$$

$$r = b - Lx, \tag{1.10}$$

$$\Delta r = \frac{||r_{\text{final}}||}{||r_{\text{initial}}||}, \tag{1.11}$$

$$\text{work} = \frac{\text{total FLOPS}}{\text{FLOPS to compute the residual on finest level}}. \tag{1.12}$$

WDA measures how much work is required to reduce the residual by an order of magnitude. r is the residual, and Δr is the change in residual norm from an initial solution to a final solution. Work is expressed in terms of the number of FLOPS required for a solve divided by that required to compute a residual on the finest level. This measure is also proportional to required memory transfers and is typically approximated in terms of the number of nonzeros in the sparse operators A_l, R_l, P_l. WDA only measures the efficiency of a single solve; it does not take the setup phase into account. WDA also does not account for parallel scalability.

1.4 Debugging Multigrid Performance

Multigrid methods are notoriously hard to write and debug. Small changes in the implementation can lead to large changes in the rate of convergence. This section discusses some methods for debugging poor

multigrid performance.

Multigrid literature contains many bounds on convergence rates [47]. Two useful bounds are the strong and weak approximation properties. The Strong Approximation Property (SAP) is defined as

$$\min_u ||e - Pu||_A^2 \leq \frac{C}{||A||} \langle Ae, Ae \rangle,$$

where e is some fine grid error and C is a constant bounding convergence speed [47, 59, 67]. Satisfying the SAP guarantees convergence for a multigrid V-cycle. The SAP gives a constraint on the modes that the coarse grid needs to solve well. Specifically, those modes where $||Ae||$ is small must be exactly reproduced by the coarse grid. In the multigrid literature, these modes are called the *near-nullspace*. Using a restriction of the near-nullspace to each aggregate in the prolongation operator is a common technique for addressing these modes.

The Weak Approximation Property (WAP) is defined as

$$\min_u ||e - Pu||^2 \leq \frac{C}{||A||} \langle Ae, e \rangle.$$

The WAP is necessary for convergence but provides a more local interpretation of what is required. Like the SAP, the WAP also indicates that near-nullspace modes need to be reproduced on the coarse grids. Both the strong and weak approximation properties do not directly provide e vectors which need to be approximated exactly. However, they can be used to compare the impact of various near-nullspace vectors if they are already known.

1.4.1 Compatible Relaxation

Compatible relaxation is a tool for finding slowly converging modes on a given level. Compatible relaxation was originally developed as a relaxation scheme for multigrid [18], but it can also be used as a technique for evaluating the effectiveness of the coarse grid correction in a multigrid scheme [27]. It can be applied to both classical and aggregation based multigrid methods. The idea is to project the coarse space out of the fine level space leaving a "smoother" space [27]. This space should be within the range quickly relaxed by the smoother. If not, the smoother either needs to be improved to cover this space or the prolongation operator needs to be improved to interpolate these modes.

Compatible relaxation provides the following bound on convergence speed of a multigrid algorithm. Given a matrix $\mathcal{S}$ which goes from $R^n->R^{n_f}$, where $n_f = n - n_c$ (n_f is the smoother space), and a smoothing matrix M, the convergence factor of our multigrid scheme will be bounded by

$$\frac{1}{\lambda_{\text{min}}\left((\mathcal{S}^T M \mathcal{S})^{-1}(\mathcal{S}^T A \mathcal{S})\right)} \text{ [27].}$$

For our problems, we either use $M = \text{diagonal}(A)$ or $M = \text{block_diagonal}(A)$. Essentially, the slowest to converge mode is the eigenvector associated with the smallest eigenvalue of the smoother applied to the smoother space variables. For the graph Laplacian problem, finding the small eigenvalues of $(\mathcal{S}^T M \mathcal{S})^{-1}(\mathcal{S}^T A \mathcal{S})$ is hard. Instead, we find the eigenvalues of $\mathcal{S}^T M^{-1} \mathcal{S} \mathcal{S}^T A \mathcal{S}$. For bundle adjustment we use $(\mathcal{S}^T M \mathcal{S})^{-1}(\mathcal{S}^T A \mathcal{S})$.

The question remains: how to construct $\mathcal{S}$? Falgout and Vassilveski present a way to construct $\mathcal{S}$ for a coarse-fine splitting [27], but we are interested in aggregation based methods. For an aggregation based method with piecewise-constant prolongation, the smoother space is modes on the aggregate that have mean zero (as piecewise constant aggregation makes the coarse level the mean of the aggregate). For non piecewise-constant prolongation, the following suffices:

$$U_{\text{agg}} V_{\text{agg}} W_{\text{agg}} = I - B_{\text{agg}} (B_{\text{agg}}^T B_{\text{agg}})^{-1} B_{\text{agg}}^T \quad \text{for every aggregate agg,} \tag{1.13}$$

$$L_{\text{agg}} = U_i \qquad \forall\, V_{\text{agg}_i} > 0, \tag{1.14}$$

$$\mathcal{S} = \begin{bmatrix} L_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & L_n \end{bmatrix}. \tag{1.15}$$

We apply the SVD to drop singular vectors in the aggregate and to orthogonalize. If we look at the smallest eigenvectors of $(\mathcal{S}^T M \mathcal{S})^{-1}(\mathcal{S}^T A \mathcal{S})$ we can get an idea of which aggregates are bad, or if we should aggregate more. If the large components of the smallest eigenvectors are limited to one aggregate, then it is likely that that aggregate should be split or not formed in the first place (as is seen in figures 1.4.1 and 1.4.1). On the other hand, if there are long range modes in the smallest eigenvectors, then the coarse space is not accurately representing the near-nullspace. For example, on the graph Laplacian, the smallest

eigenvectors often have most of their magnitude in a single aggregate. This can inform a splitting of the aggregate: the sign of the vector indicates which resulting aggregate each dof should be put in. Looking at such splitting can inform the choice of heuristics or improvements for aggregation. Furthermore, if we see a clustering of small eigenvalues, it could indicate that there is a fundamental flaw in how vertices are aggregated.

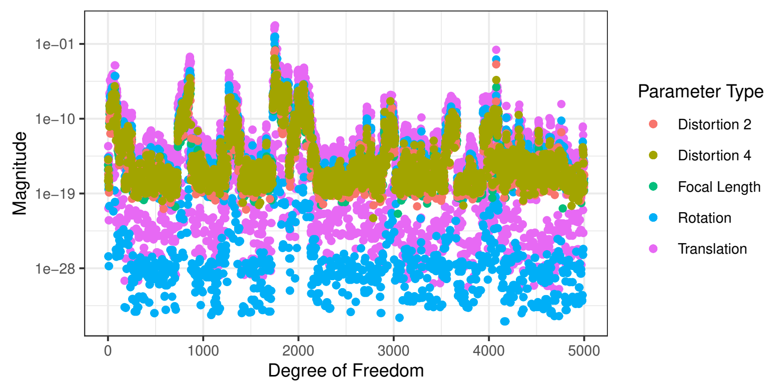


Figure 1.1: Plot of magnitude of each component in the smallest compatible relaxation eigenvector of a bundle adjustment linear system. Only a few components in the vector have a large magnitude, indicating there are no long range effects present.

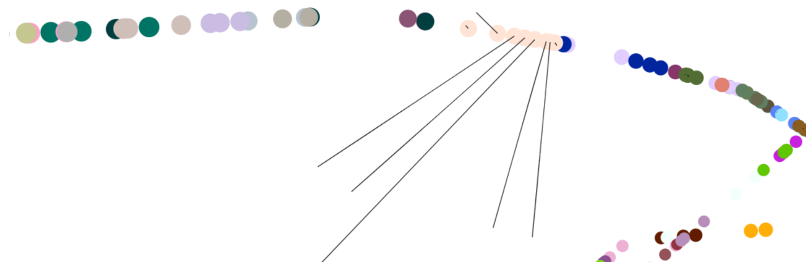


Figure 1.2: Plot of smallest eigenvector components on a per degree of freedom basis in a bundle adjustment linear system. Color of points indicates which aggregate they belong to. Black lines indicate direction of movement of the dof according to the smallest eigenvector. All of the large components are in a single aggregate, indicating that this aggregate should either be split or a stronger smoother should be used. There is no clear partition of the aggregate using the small eigenvector components, leading us to believe that a stronger smoother is needed.

1.5 Distributed Memory Parallelism

Problems in both Graph Laplacians and Bundle Adjustment can be very large—up to billions of nonzeros. At this scale, problems can no longer fit in memory on a single computer, or if they can, solve times take too long. To be able to solve larger problems, multi-process distributed memory environments is a must. Distributed memory parallelism is normally used on clusters of computers connected via a high speed link. In this environment, the problem is split across many processes of the cluster—each process is given a subset of the degrees of freedom of the whole problem. In distributed linear algebra, the communication is block synchronous: each process computes on its local block of the matrix and then sends its portion of the computation to relevant other processes. Writing distributed memory algorithms is challenging as they require thought in how the data is laid out between processes and how the communication between processes is structured. The data needs to be laid out in such a way that each process has about the same amount of work to do. If there is too much of a difference, then the processes with less work spend most of their time waiting for the processes with more work. This is called *load imbalance*.

In distributed memory environments scaling properties of techniques are important. Ideally, as more processes are used, solution time will decrease proportionally. However, in most cases, solution time will scale less than optimally. We will try to measure how close our algorithms come to optimality. There are two standard ways of measuring scalability: weak and strong scaling. In weak scaling the amount of work per process is held constant while the number of processes increases. Weak scaling is useful in approximating the amount of inherent serial work in a problem. Strong scaling maintains a constant total amount of work and varies the number of processes. Strong scaling is useful in evaluating the number of processes that should be used for given problem to get the desired time to solution.

Chapter 2

LigMG — A Parallel Graph Laplacian Solver

This chapter is finished work accepted at PASC '18 done in collaboration with Jed Brown [38]. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0016140. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

2.1 Related Work

A variety of different solvers that have been proposed for solving large sparse graph Laplacian systems. Some are general purpose solvers, whereas others are tailored specifically for graph Laplacians.

2.1.1 Direct Solvers

Direct solvers can be a good choice for solving systems to a high accuracy. Some direct solvers for sparse systems, such as SuperLU_DIST [43] and MUMPS [6], function in distributed memory. However, none of these solvers performs well on large graph Laplacian systems because "small" vertex separators [30] do not exist. Because we are interested in very large systems, we require that our solver scales linearly in the number of nonzeros in the matrix, which these direct solvers do not.

2.1.2 Simpler Preconditioners

Iterative solvers such as conjugate gradients, coupled with simple preconditioners like Jacobi or incomplete Cholesky, sometimes perform well on highly irregular graphs. For example, Jacobi (diagonal) preconditioning is often sufficient for social network graphs with small diameter and Incomplete Cholesky may provide more robustness, but tends to exhibit poor parallel scalability.

2.1.3 Theoretical Solvers

A variety of theoretical Laplacian solvers have been proposed in literature starting with Spielman and Teng's 2003 paper [65]. To our knowledge, no working implementation of this algorithm exists. Since then, many more theoretical linear solvers have been proposed. Most use either a support graph or low stretch spanning tree sparsifier as a preconditioner for a Krylov iterative solver. Several serial implementations of these ideas exist in "Laplacians.jl" a package written by Daniel Spielman [64].

Kelner et al. later proposed a novel technique with a complexity of $O(m \log^2 n \log \log n \log (\epsilon^{-1}))$ [34] (n is the number of vertices, m is the number of edges, ϵ is the solution tolerance). An implementation of this algorithm exists but appears to not be practical (as of yet) [16].

2.1.4 Practical Serial Solvers

Three practical graph Laplacian solvers have been proposed: Koutis and Miller's Combinatorial Multigrid (CMG) [39], Livne and Brandt's Lean Algebraic Multigrid (LAMG) [46], and Napov and Notay's Degree-aware Rooted Aggregation (DRA) [49]. All use multigrid techniques to solve the Laplacian problem.

Combinatorial Multigrid, like much of the theoretical literature, takes a graph theoretic approach. It constructs a multilevel preconditioner using clustering on a modified spanning tree [39]. Its main focus is on problems arising in imaging applications. The spanning tree construction and clustering presented in CMG do not lend themselves to a simple parallel implementation.

Lean Algebraic Multigrid uses a more standard AMG approach with modifications suited for Laplacian matrices. Notably, it employs unsmoothed aggregation tailored to scale-free graphs, a specialized distance

function, and a Krylov method to accelerate solutions on each of the multigrid levels. These changes are not rooted in graph theory but produce good empirical results. Empirically, LAMG is slightly slower than CMG but more robust [46]. LAMG's partial elimination procedure and clustering process are both inherently serial.

Degree-aware Rooted Aggregation applies a similar partial elimination technique to LAMG, except it is limited to degree 1 vertices. Its performance relies on a combination of unsmoothed aggregation based on vertex degree and a multilevel Krylov method called K-cycles[49]. A K-cycle is a multigrid W-cycle (one can imagine a K-cycle with different cycle index, but we only consider K-cycles with a cycle index of 2) with Krylov acceleration applied at each level. DRA's aggregation (like LAMG's) is inherently serial and would require modifications for parallelism.

2.2 Main Contribution

Our solver uses an unsmoothed aggregation based multigrid technique with low degree elimination. Its notable features are: 1. a matrix distribution that improves parallel performance but increases complexity of aggregation and elimination algorithms 2. a parallel elimination algorithm using this matrix distribution 3. a parallel aggregation algorithm also using this matrix distribution. We started building our solver by analyzing the performance of LAMG and its potential for parallelism.

2.2.1 Issues With a Parallel Implementation of LAMG

To understand what parts of LAMG we could adapt to a distributed memory setting, we ran LAMG on 110 graphs from the University of Florida Sparse Matrix Collection [25] while varying its parameters for cycle index, smoother and iterate recombination (we would like to also vary the aggregation routine, but LAMG's energy *unaware* aggregation is not working in the MATLAB implementation). LAMG uses a cycle index of 1.5, which is half way between a V-cycle and a W-cycle. Iterate recombination is a multilevel Krylov method that chooses the optimal search direction from the previous solution guesses at each level. It is similar in nature to K-cycles (see section 2.2.7). Our goal was to find which parts of LAMG had the largest

effect on solver performance.

Figure 2.1 shows the result of these tests. Visually, Gauss Seidel smoothing and iterate recombination together give the smallest WDA. Cycle index has a very small effect relative to smoother and iterate recombination.

We applied a statistical analysis to LAMG's performance with its features on and off. The largest change in solver performance came from using Gauss-Seidel smoothing over Jacobi smoothing. On social network graphs, Gauss-Seidel smoothing often performs much better than Jacobi smoothing. Iterate recombination is the second most important factor. It helps improve performance on high diameter graphs such as road networks. The least important factor is the 1.5 cycle index. Although it does slightly improve WDA, it is much less important than Gauss-Seidel smoothing and iterate recombination.

Given the importance of Gauss-Seidel smoothing and iterate recombination, we would like to use them in a parallel implementation. However, there are a couple of challenges with using these features in a parallel implementation of LAMG:

(1) The power-law vertex degree distribution can cause large work and communication imbalances.

(2) LAMG's low degree elimination is a sequential algorithm.

(3) LAMG's energy-based aggregation is a sequential algorithm.

(4) Multilevel Krylov acceleration can be a parallel bottleneck.

(5) W-cycles exhibit poor parallel performance.

(6) Gauss-Seidel smoothing is infeasible in parallel.

2.2.2 2D Matrix Distribution

Our initial implementation used a vertex distribution of the graph: each processor owns some number of rows in the Laplacian matrix. This initial implementation scaled very poorly as we increased the number of processes. One process had many more edges than all the other processes, causing a work imbalance. Using a partitioner can temporarily alleviate this problem, but in the limiting case (where each process

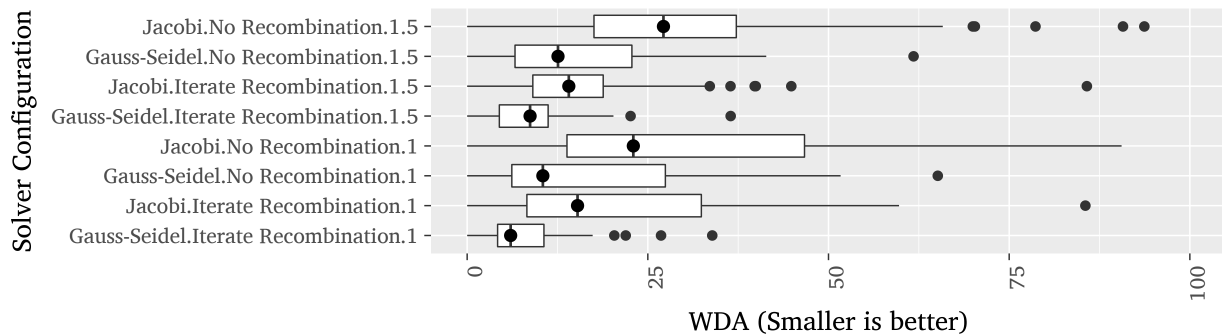


Figure 2.1: Boxplots of the performance of serial LAMG [45] on 110 graphs from the University of Florida Sparse Matrix collection [25]. Solver configuration (vertical axis) is a triple of smoother, iterate recombination (or not), and cycle index. Performance is measured in terms of work per digit of accuracy (see section 1.3.4). WDA accounts for work per iteration and number of iterations. Problems are solved to a relative tolerance of 10^{-8}. Each box represents the interquartile range of WDA for a given solver configuration. The line and dot inside the box indicates the median WDA. Horizontal lines on either side of the box indicate the range of WDA values. Dots outside the box indicate outliers.

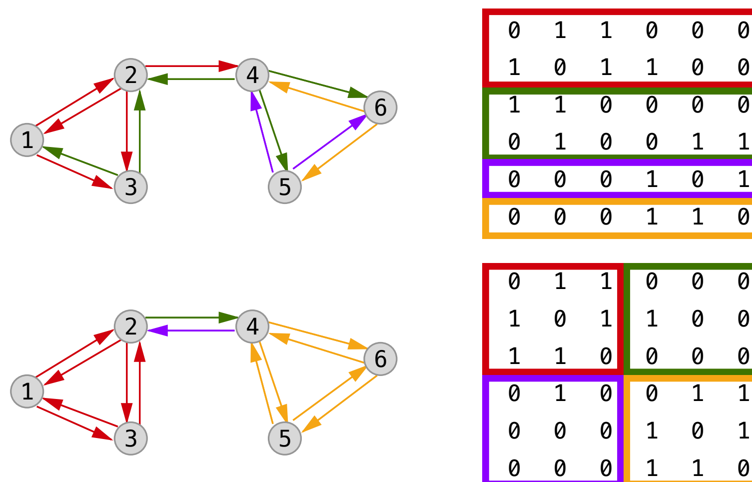


Figure 2.2: The distribution of directed edges in the graph (left) and adjacency matrix (right). The top is a 1D vertex distribution, and the bottom is a 2D edge distribution. Each color corresponds to edges and matrix entries owned by processor. Note that no process in the 2D distribution has all the out-edges or in-edges for a given vertex.

owns a single vertex), processes with hubs will have significantly more work than those without. Research on scaling sparse matrix operations on adjacency matrices suggests that a more sophisticated distribution of matrix entries is important. We use CombBLAS, which has demonstrated the scalability and load balancing benefits of a 2D matrix distribution [21]. A 2D matrix distribution can be thought of as a partition of graph edges instead of vertices. Computational nodes (or processes) form a 2D grid over the matrix (called a processor grid). Each process is given a block of the matrix corresponding to its position in the grid (see Figure 2.2). Vectors can either be distributed across all processes or just processes on the diagonal. In our implementation, we found performance did not change with vector distribution, so we distribute them across diagonal entries in the processor grid. The disadvantage of a 2D distribution is that it has higher constant factors and poorer data locality.

CombBLAS expresses graph algorithms in the language of linear algebra. Instead of the usual multiplication and addition used in matrix products, CombBLAS allows the user to use custom multiplication and addition operations. We will follow the $\oplus.\otimes$ notation used by GraphBLAS (the standardization effort of CombBLAS)[35]. Here, $\oplus$ specifies the custom addition operator and $\otimes$ the multiplication. While an $m \times n$ matrix A maps from $\mathbb{R}^n$ to $\mathbb{R}^m$, a generalized matrix $\hat{A}$ maps from $\mathbb{C}^n$ to $\mathbb{D}^m$, where $\mathbb{C}$ and $\mathbb{D}$ may be different.

$$(Av)_i \coloneqq \sum_j A_{ij} v_j, \qquad \text{Usual matrix product} \tag{2.1}$$

$$(\hat{A} \oplus.\otimes v)_i \coloneqq \bigoplus_j \hat{A}_{ij} \otimes v_j, \qquad \text{Generalized matrix product} \tag{2.2}$$

$$\hat{A} \in \mathbb{B}^{m \times n}, \quad v \in \mathbb{C}^n, \tag{2.3}$$

$$\otimes \colon \mathbb{B} \times \mathbb{C} \to \mathbb{D}, \quad \oplus \colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}. \tag{2.4}$$

Our operators are similar in structure to a semiring, but they permit different types of elements in the input matrix and the input and output vectors. If we structure our algorithms in terms of generalized matrix-vector products, we can piggyback on the proven performance of CombBLAS. However, if we cannot express all parts of an algorithm using this linear algebraic approach, we must keep in mind that no computational node has a complete view of all the edges to and from any vertex. Implementing an arbi-

trary vertex neighborhood operation, such as choosing the median of neighbors, would require potentially non-scalable custom communication.

2.2.3 Random Vertex Ordering

A 2D matrix distribution alleviates communication bottlenecks and some load balancing difficulties, but the processes responsible for diagonal blocks are often found to have many more nonzeros than typical off-diagonal blocks. For example, social network often have a couple of large degree "hubs" that are connected to many other vertices. These vertices correspond to an almost dense column and row in the graph Laplacian. Often a single process will end up with a few hubs and have 10x (or more) edges than other processes. A simple technique to better balance the workload is to randomly order vertices. This trades data locality for better load distribution. More sophisticated techniques exist for 2D matrix partitioning [15], but we found that a random distribution is sufficient for acceptable load balance. We found that random vertex ordering increased not only asymptotic parallel scalability but also performance for relatively small process counts. We apply this randomization only to the input matrix; we do not re-randomize at coarser levels.

2.2.4 Parallel Low-Degree Elimination

Low-degree elimination greatly reduces problem complexity in graph Laplacians, especially those arising from social networks. Like LAMG, we eliminate vertices of degree 4 or less.

The main difficulty in adapting low-degree elimination to a distributed memory system is deciding which vertices to eliminate. If we had a vertex centric distribution, each process could locally decide which of its local vertices to eliminate. However, we have a 2D edge distribution, so we will instead structure our elimination in terms of linear algebra and allow CombBLAS to do the heavy lifting.

Our algorithm for low-degree elimination is detailed in Algorithm 3. It essentially boils down to two steps. First, mark all vertices of degree 4 or less as candidates for elimination. Then, for each candidate, check whether it has the lowest hash value among all neighboring candidates. If it has the lowest hash value, it will be eliminated. The hash value is a hash of the vertex's id. We use a hash of the id instead of

the id itself in order to prevent biases that might occur when using a non-random matrix ordering.

In linear algebraic terms, choosing vertices to eliminate can be accomplished by creating a vector that marks each vertex as a candidate or not (line 2 of Algorithm 3) and then multiplying said vector with the Laplacian matrix using a custom $\otimes$ and $\oplus$ (line 2). The $\otimes_{\text{elim}}$ filters out matrix entries that are not candidates and neighbors:

$$a \otimes_{\text{elim}} c = \begin{cases} c, & \text{if } a \neq 0, \\ \varnothing, & \text{otherwise,} \end{cases}$$

where $c = \varnothing$ is used to indicate that a vertex should not be considered.

The $\oplus_{\text{elim}}$ chooses the candidate with the smallest hashed id,

$$x \oplus_{\text{elim}} y = \begin{cases} x, & \text{if } \text{hash}(x) \leq \text{hash}(y), \\ y, & \text{if } \text{hash}(x) > \text{hash}(y), \end{cases}$$

where $\text{hash}(\varnothing) = \infty$.

Algorithm 3 Determine vertices to eliminate

1: **function** LOW-DEGREE ELIMINATION($L \in \mathbb{R}^{n \times n}$)
2: *candidates*$_i$ ← i **if** $\text{degree}(V_i) \leq 4$ **else** $\varnothing$, $\quad i \in V$
3: $z \leftarrow L \oplus_{\text{elim}} . \otimes_{\text{elim}}$ *candidates*
4: **if** $z_i = i$ **then** eliminate V_i
5: **end function**

We use the Laplacian L so that the neighborhood of each vertex contains itself. Each entry z_i corresponds to the neighbor of v_i that is a candidate and has the lowest hashed id. If $z_i = i$, then we know that v_i is the candidate with the smallest hash among its neighbors (line 4) and can be eliminated. Let $\mathcal{F}$ denote eliminated vertices and $\mathcal{C}$ be vertices that have not been eliminated. Following Livne and Brandt [46], we express elimination in the language of multigrid. We first introduce a permutation Π of the degrees of freedom such that

$$L_l = \Pi \begin{pmatrix} L_{\mathcal{F}\mathcal{F}} & L_{\mathcal{F}\mathcal{C}} \\ L_{\mathcal{F}\mathcal{C}}^T & L_{\mathcal{C}\mathcal{C}} \end{pmatrix} \Pi^T,$$

which admits the block factorization

$$L_l = \Pi \begin{pmatrix} I & \\ L_{\mathcal{FC}}^T L_{\mathcal{FF}}^{-1} & I \end{pmatrix} \begin{pmatrix} L_{\mathcal{FF}} & \\ & L_{l+1} \end{pmatrix} \begin{pmatrix} I & L_{\mathcal{FF}}^{-1} L_{\mathcal{FC}} \\ 0 & I \end{pmatrix} \Pi^T$$

in terms of the Schur complement

$$L_{l+1} = L_{\mathcal{CC}} - L_{\mathcal{FC}}^T L_{\mathcal{FF}}^{-1} L_{\mathcal{FC}}.$$

Note that $L_{\mathcal{FF}}$ is diagonal so its inverse is also diagonal and that we can alternately express $L_{l+1} = P_l^T L_l P_l$ in terms of the prolongation

$$P_l = \Pi \begin{pmatrix} -L_{\mathcal{FF}}^{-1} L_{\mathcal{FC}} \\ I \end{pmatrix}.$$

Inverting the block factorization yields

$$\begin{aligned} L_l^{-1} &= \Pi \begin{pmatrix} I & -L_{\mathcal{FF}}^{-1} L_{\mathcal{FC}} \\ 0 & I \end{pmatrix} \begin{pmatrix} L_{\mathcal{FF}}^{-1} & \\ & L_{l+1}^{-1} \end{pmatrix} \begin{pmatrix} I & \\ -L_{\mathcal{FC}}^T L_{\mathcal{FF}}^{-1} & I \end{pmatrix} \Pi^T, \\ &= P L_{l+1}^{-1} P^T + \Pi \begin{pmatrix} L_{\mathcal{FF}}^{-1} & 0 \\ 0 & 0 \end{pmatrix} \Pi^T, \end{aligned}$$

which is an additive 2-level method with $\mathcal{F}$-point smoother

$$\mathcal{F}\text{smooth}(x, b) = \Pi \begin{pmatrix} L_{\mathcal{FF}}^{-1} & 0 \\ 0 & 0 \end{pmatrix} \Pi^T b.$$

Rather than applying L_{l+1}^{-1} exactly, we approximate it by continuing the multigrid cycle.

The ids of eliminated vertices are broadcast down processor rows and columns. Each process constructs entries of $L_{\mathcal{FC}}$ and $L_{\mathcal{FF}}^{-1}$ that depend on its local entries in L_l. These constructed entries are then scattered to the processes that own them in P_l. Alternatively, we could construct Π, $L_{\mathcal{FC}}$, and $L_{\mathcal{FF}}^{-1}$ explicitly and use them to build P_l.

Our candidate selection scheme is not as powerful as the serial LAMG scheme. The serial scheme will eliminate every other vertex of a chain. In the best case we do the same, but in the worst case we eliminate only one vertex if the hash values of vertices in the chain are in sequential order. To address this issue, we

can run low-degree elimination multiple times in a row to eliminate more of the graph. In practice, we find one iteration is sufficient to remove most of the low degree structure.

We apply low-degree elimination before every aggregation level and only if more than 5% of the vertices will be eliminated.

2.2.5 Parallel Aggregation

Our parallel aggregation algorithm (Algorithm 4) uses a strength-of-connection metric, S, to determine how to form aggregates. It indirectly determines how likely any two vertices will be clustered together. We use the affinity strength-of-connection metric proposed by Livne and Brandt in the LAMG paper [46]. To construct the strength-of-connection matrix, S, we smooth four vectors three times each. The relevant parts of the smoothed vectors are broadcast down communication grid rows and columns. Each process then constructs its local part of the matrix using the affinity metric:

$$L \in \mathbb{R}^{n\times n}, y \in \mathbb{R}^{n\times m}, m \ll n, \text{random entries}, \tag{2.5}$$

$$x := \text{smooth on } Ly = 0, \tag{2.6}$$

$$C_{ij} := \begin{cases} 0, & \text{if } A_{ij} = 0 \text{ or } i = j, \\ \dfrac{|\sum_{k=1}^{m} x_{ik}x_{jk}|^2}{(\sum_{k=1}^{m} x_{ik}x_{ik})^2(\sum_{k=1}^{m} x_{jk}x_{jk})^2}, & \text{otherwise}, \end{cases} \tag{2.7}$$

$$S_{ij} := \frac{C_{ij}}{\max\left(\max_{s\neq i} C_{is}, \max_{s\neq j} C_{s,j}\right)}, \tag{2.8}$$

where A is the adjacency matrix. The smoothing we use is three iterations of Jacobi smoothing. The total cost of creating S is 4 vectors $*$ 3 smoothing iterations, for 12 matrix-vectors multiplies total. Currently, we smooth each vector separately and do not exploit any of the parallelism available in smoothing multiple vectors together.

The construction of C_{ij} is entirely local because C has the same distribution as A. Constructing S requires communication along processor rows and columns to find the largest nonzero of C for each column and row in the matrix. Note that S has 0 diagonal.

Our aggregation algorithm uses a voting scheme in which each vertex votes for which of its neighbors it would like to aggregate with. A vector $status$ contains a state $\in \{\textit{Seed}, \textit{Undecided}, \textit{Decided}\}$ and an index

Algorithm 4 Aggregation

1: **function** AGGREGATION($S \in \mathbb{R}^{n \times n}$)
2: $\quad$ $status_i \leftarrow (Undecided, i)$, **for** i **in** $1..n$
3: $\quad$ $votes_i \leftarrow 0$, **for** i **in** $1..n$
4: $\quad$ **for** *iter* **in** $1..10$ **do**
5: $\quad\quad$ $status, votes \leftarrow$ AGGREGATION-STEP($S, status, votes, 0.5^{iter}$)
6: $\quad$ **end for**
7: $\quad$ **for** i **in** $1..n$ **do**
8: $\quad\quad$ $(\cdot, j) \leftarrow status_i$
9: $\quad\quad$ $aggregates_i \leftarrow j$
10: $\quad$ **end for**
11: $\quad$ **return** *aggregates*
12: **end function**
13: **function** AGGREGATION-STEP($S, status, votes, filter\text{-}factor$)
$\triangleright$ *status* is a vector with elements of type $(State, Index)$
$\triangleright$ S is the strength-of-connection matrix
14: $\quad$ $S_{filt} \leftarrow$ Remove nonzeros $<$ *filter-factor* from S
15: $\quad$ $d \leftarrow S_{filt} \oplus_{\text{agg}}.\otimes_{\text{agg}}\, status$
16: $\quad$ *local_votes* $\leftarrow$ Sparse map containing votes for vertices
17: $\quad$ **for** i **in** $1..n$ **do**
18: $\quad\quad$ $(s, j, w) \leftarrow d_i$
19: $\quad\quad$ **if** $s = Seed$ **then**
20: $\quad\quad\quad$ $status_i \leftarrow (Decided, j)$ $\triangleright$ Found a neighboring seed, V_i is aggregated with V_n
21: $\quad\quad$ **else if** $s = Undecided$ **then**
22: $\quad\quad\quad$ $local_votes[j] \leftarrow local_votes[j] + 1$ $\triangleright$ No neighboring seed, V_i votes for V_n
23: $\quad\quad$ **end if**
24: $\quad$ **end for**
25: $\quad$ *local_votes* $\leftarrow$ reduce_by_key$(+, local_votes)$ $\triangleright$ Communicate local votes
26: $\quad$ $votes \leftarrow votes + local_votes$ $\triangleright$ Update persistent votes counts
27: $\quad$ **for** i **in** $1..n$ **do**
28: $\quad\quad$ **if** $votes_i > 8$ & $status_i = (Undecided, i)$ **then**
29: $\quad\quad\quad$ $status_i \leftarrow (Seed, i)$ $\triangleright$ Vertices with enough votes become *Seeds*
30: $\quad\quad$ **end if**
31: $\quad$ **end for**
32: $\quad$ **return** *status*, *votes*, *aggregates*
33: **end function**

for each vertex,

$$\textit{status}_i := \begin{cases} (\textit{Seed}, i), & \text{Vertex } i \text{ is a } \textit{Seed}, \\ (\textit{Undecided}, i), & \text{Vertex } i \text{ has not yet joined an aggregate}, \\ (\textit{Decided}, j), & \text{Vertex } i \text{ is aggregated with } \textit{Seed } j. \end{cases}$$

Initially, $\textit{status}_i = (\textit{Undecided}, i)$. In each voting iteration, each *Undecided* vertex either aggregates with a neighboring *Seed* (and becomes *Decided*) or votes for a neighboring *Undecided* vertex to become a *Seed*. If a vertex is voted for enough times, it will turn into a *Seed*. The strength-of-connection matrix determines which neighboring vertex is aggregated to or voted for. For the first round of aggregation, we only consider very strong connections (> 0.5). We gradually reduce this bound in subsequent iterations (Livne and Brandt apply a similar technique in LAMG [46]). The vertices' choice of neighbor is expressed as a matrix-vector product $S \oplus_{\text{agg}} . \otimes_{\text{agg}} \textit{status}$ with

$$w \otimes_{\text{agg}} (\textit{state}, i) := \begin{cases} (\textit{state}, i, w), & \text{if } w \neq 0, \\ (\textit{Decided}, -1, 0), & \text{otherwise}, \end{cases} \tag{2.9}$$

$$(\textit{state}_a, i_a, w_a) \oplus_{\text{agg}} (\textit{state}_b, i_b, w_b) := \begin{cases} (\textit{state}_a, i_a, w_a), & \text{if } \textit{state}_a = \textit{state}_b \;\&\; w_a \geq w_b, \\ (\textit{state}_b, i_b, w_b), & \text{if } \textit{state}_a = \textit{state}_b \;\&\; w_a < w_b, \\ (\textit{state}_a, i_a, w_a), & \text{if } \textit{state}_a > \textit{state}_b, \\ (\textit{state}_b, i_b, w_b), & \text{if } \textit{state}_a < \textit{state}_b, \end{cases} \tag{2.10}$$

where *Seed* $>$ *Undecided* $>$ *Decided*. Note that the input vector contains pairs, whereas the output vector contains 3-tuples.

The votes for each vertex are tallied using a sparse reduction that has the same communication structure as a matrix-vector product. Our implementation uses an allreduce because it has lower constants and is not a bottleneck. Using the tallied votes, we update the status vector with new roots and aggregates. The vote counts are persisted across voting iterations so that vertices with low degree may eventually become seeds. We choose to do 10 voting iterations, and we convert *Undecided* vertices to *Seeds* if they receive 8 or more

votes. Both these numbers are arbitrary. In practice, we find that performance is not sensitive to increasing or decreasing these constants by moderate amounts.

The result of aggregation is a distributed vector v, where vertex i is part of aggregate v_i. We perform a global reordering so that aggregates are numbered starting at 0. We construct R by inserting $R_{v_j j} = 1$, where j is in the locally owned portion of v, then scattering to a balanced 2D distribution. This 2D distribution is similar to L, except each process has a rectangular local block instead of a square one.

$$R_{ij} := \begin{cases} 1, & \text{if } v_j = i \ (V_j \text{ is in aggregate } i), \\ 0, & \text{otherwise.} \end{cases} \tag{2.11}$$

$$P, := R^T, \quad L_{l+1} := R_l L_l P_l. \tag{2.12}$$

2.2.6 Smoothing

In general, Gauss-Seidel smoothing is the best performing smoother on graph Laplacians (section 2.2.1 provides more details on Gauss-Seidel vs Jacobi performance). However, its parallel performance on graph Laplacians is very poor. Most processes have an overwhelming amount of connections that reference values outside of the local block of the matrix, and the graph cannot be colored with a reasonable number of colors. Our resulting choice of smoother is Chebyshev/Jacobi smoothing [2] because it is stronger than (weighted) Jacobi with equivalent parallel performance. Instead of applying k iterations of a smoother, we use one application of degree k Chebyshev smoothing. We choose our lower and upper bounds because $.3$ and 1.1 times the largest eigenvalue as estimated by 10 Arnoldi iterations [7] (we include these iterations in our setup cost).

2.2.7 K-cycles

We would like to include some form of multilevel Krylov acceleration because it improves solver robustness (see section 2.2.1 for more details). We implemented K-cycles as described by Notay and Vassilevski in “Recursive Krylov-based multigrid cycles” [55] and used in Napov and Notay’s DRA [49]. At each aggregation level in our multigrid hierarchy, we perform a number of Flexible Conjugate Gradient (FCG)

[52] iterations using the rest of the hierarchy as a preconditioner. We do not apply Krylov acceleration to elimination levels because they exactly interpolate the solution from the coarse grid.

2.3 Numerical Results

Our solver uses a V-cycle or K-cycle with one iteration of degree 2 Chebyshev smoothing before restriction and one iteration of degree 2 Chebyshev smoothing after prolongation. The V-cycle is used as a preconditioner for Conjugate Gradient (FCG when using K-cycles). Our solver is implemented in C++ and uses CombBLAS [21] for sparse linear algebra and PETSc [10, 11] for Chebyshev smoothing, eigenvalue estimation, and Krylov methods.

Our numerical tests were run on NERSC's Edison and Cori clusters. Edison is a Cray XC30 supercomputer with 24 "Ivy Bridge" Intel Xeon E5-2695 v2 cores per node and a Cray Aries interconnect. Cori is Cray supercomputer with 36 "Haswell" Intel Xeon E5-2698 v3 cores per node and a Cray Aries interconnect. For each test, we run four MPI processes per physical node in order to obtain close to peak bandwidth.

We solve to a relative tolerance of 10^{-8}. We use a random right hand side with the constant vector projected out. We also tested with a right hand side composed of low eigenmodes but did not notice any difference in performance compared to a random right hand side. The coarsest level size is set to have no more than 1000 nonzeros in L. Our solver uses somewhere from 20 to 40 levels depending on the problem. In order to ensure enough work per process, we use a smaller number of processors on coarser levels if the amount of work drops below a threshold (if nnz$(L)/10000 <$ number of processors). The coarsest level always ends up on a single process.

2.3.1 Comparison to Serial

We compared our parallel solver to Livne and Brandt's serial LAMG implementation in MATLAB [45, 46]. It is hard to fairly compare single threaded performance between our solver and LAMG. One or the other could be better optimized, or choice of programming language might make a difference. To provide a fair comparison, we measure the work per digit of accuracy of each solver.

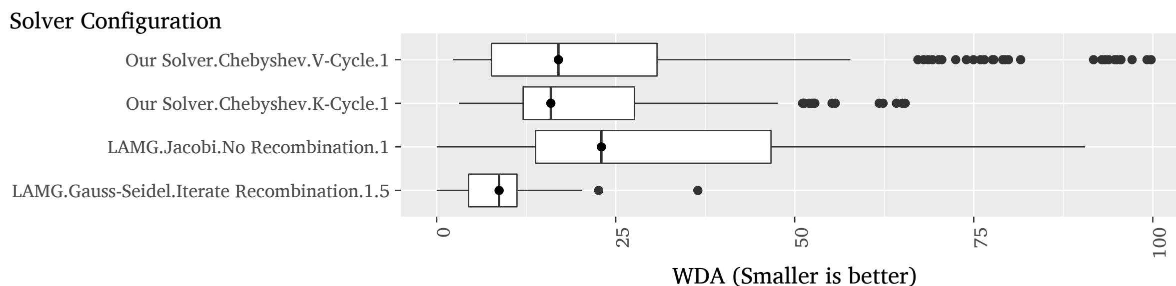


Figure 2.3: Boxplots of solver performance in various configurations on a selection of 110 graphs from the University of Florida Sparse Matrix Collection [25]. Performance is measured in terms of work per digit of accuracy (see section 1.3.4). WDA accounts for work per iteration and number of iterations. The last number in the solver configuration indicates the cycle index. All solves to a relative tolerance of 10^{-8}. LAMG with Jacobi smoothing, no recombination, and cycle index 1 has many undisplayed outliers because they fall well above 100 WDA.

In general, our method exhibited poorer convergence factors than the serial LAMG implementation. We expect that our solver performs worse because we have made multiple concessions for parallel scalability. These changes are:

(1) No energy-based aggregation

(2) Chebyshev smoothing (versus Gauss-Seidel)

(3) No 1.5 cycle index

(4) No multilevel Krylov acceleration (for V-cycles)

Figure 2.3 shows the performance (measured in terms of WDA) of serial LAMG and our solver. The fourth line shows LAMG with all of its parallel-unfriendly features enabled. The third line shows LAMG without these features (but using LAMG's standard aggregation and elimination). The first line is our solver without K-cycles and the second line with K-cycles. Our solver has a higher median WDA and variance than LAMG with all features enabled. Our solver is not as robust and has more outliers. Most of these outliers are road networks. A couple of graphs have a fairly high WDA with our solver but are solved quickly by LAMG. This is expected because we have made concessions in order to achieve parallel performance. However, our solver performs much better than LAMG with parallel friendly features (Jacobi smoothing and no recombination).

Also interesting to note is the small difference in WDA of our solver with and without K-cycles. K-cycles have low variance, and hence are more robust, but median performance is not much improved. However, this small gain in WDA is not worth the high parallel cost of K-cycles (as seen in Figure 2.4). On the *hollywood* graph, K-cycles are clearly slower and scale worse than V-cycles. K-cycles need inner products on every level of the cycle (as with Krylov smoothers), which require more communication. The W-cycle structure of K-cycles also causes a parallel bottleneck. The coarsest level is visited $2^{10} - 2^{11}$ times, which results in lots of sequential solves and data redistributions. Because K-cycles appear worse than V-cycles (especially when compared to the marginal robustness they provide), we use V-cycles by default in our solver. All following performance results for our solver use V-cycles.

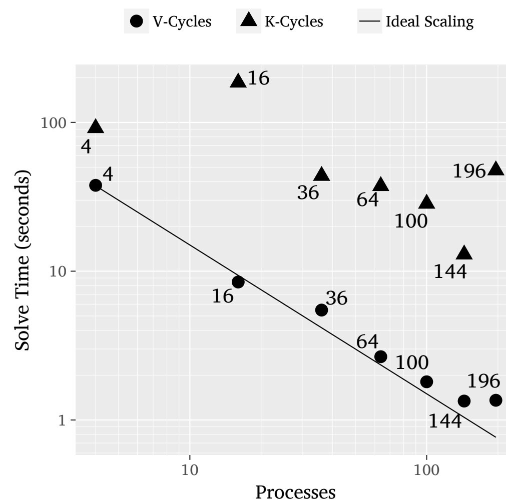


Figure 2.4: Loglog plot of strong scaling of our solver with K-cycles and with V-cycles on the *hollywood* graph (1,139,905 vertices, 113,891,327 edges) on Edison. Numeric labels next to points indicate number of processes for a given solve. There are 21-23 multigrid levels so the coarsest level is visited $2^{10} - 2^{11}$ times with an index 2 cycle. The coarse level solves and redistribution become a parallel bottleneck.

2.3.2 Strong Scaling

To measure the scalability of our approach, we measured strong scaling on four real world social network graphs using up to 576 processors. We choose these four graphs because they are some of the largest real world irregular graphs that we can find and are infeasible to solve on a single process. Many of the graphs in the 110 we use for serial experiments are small enough that solving them on a single process is fast enough.

Our largest graph, *com-friendster*, (from the Stanford Large Network Dataset Collection [41, 75]) has 3.6 billion nonzero entries in its Laplacian matrix. Solving a Laplacian of this size on a single node is infeasible (even if it could fit in memory, the solve time would be much too long). Figure 2.5 shows the efficiency, measured as

$$\frac{\mathrm{nnz}(L)}{\text{TDA} \cdot \text{number of processes}}, \quad \text{TDA} := \frac{-\text{time}}{\log_{10} \Delta r},$$

versus solve time (where TDA is time per digit of accuracy). We do not use a percentage for efficiency because the choice of base efficiency makes it difficult to compare different solves on different graphs. A horizontal line indicates that the solver is scaling optimally (increasing machine size moves to the left). The largest graph (*com-friendster*) appears to have better scaling than the smaller graphs. *com-orkut* and *hollywood* have a much higher efficiency than *com-friendster* and *com-lj* because *com-orkut* and *hollywood* have a lower WDA. If we normalize by WDA on both axes, so the y axis becomes

$$\frac{\mathrm{nnz}(L)}{\frac{\text{solve time}}{\text{work unit}} \cdot \text{number of processes}},$$

(this is similar to a "per iteration" metric), then we get Figure 2.6. Normalizing by WDA compares efficiency independent of how difficult it is to solve each graph. The smaller graphs are solved faster but scale poorly with increasing number of processes. *com-friendster* takes longer to solve because it is larger and efficiency is somewhat lower (likely due to poor cache behavior with the random ordering) but scales better than the smaller graphs. The poor scaling for smaller graphs is explained by having less work per process, resulting in a solve phase dominated by communication.

Our high setup cost (relative to solve time) also increases the number of repeated solves necessary. The majority of the setup phase is spent in the finest level triple product ($P^T LP$), which is handled by

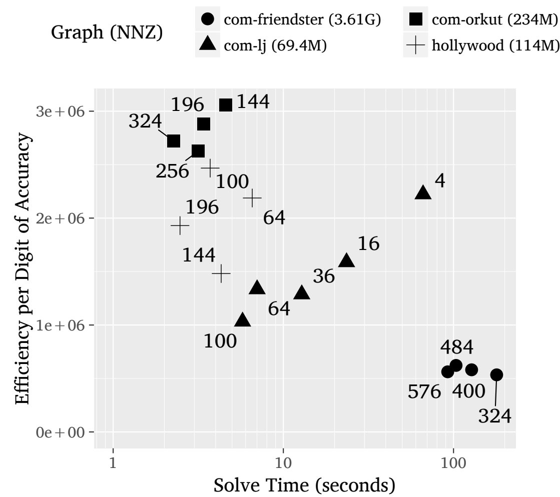


Figure 2.5: Semilog-x plot of efficiency $\left(\frac{\text{nnz}(L)}{\text{TDA}\cdot\text{number of processes}}\right)$ vs. solve time for a variety of large social network graphs on Cori. Solves are to a relative tolerance of 10^{-8}. Numeric labels next to points indicate number of processes for a given solve. *hollywood* took 15 iterations on 196 processors versus 13 iterations on all other processor sizes, leading to loss of efficiency and negligible speedup. Some solves on the same problem perform more iterations (and solve to a slightly higher tolerance) than others causing variation in efficiency.

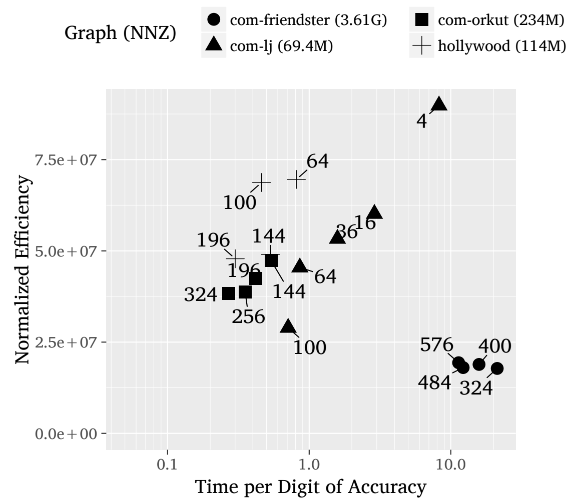


Figure 2.6: Semilog-x plot of normalized efficiency $\left(\frac{\text{nnz}(L)}{\text{time per work unit}\cdot\text{number of processes}}\right)$ vs time per digit of accuracy for a variety of large social network graphs on Cori. Numeric labels next to points indicate number of processors used for a given solve.

a generic CombBLAS matrix-matrix product using $+.*$. A matrix-matrix product that exploits the special structure of R and P would lower setup times. Still, these setup times are reasonable and can be amortized over multiple solve phases (when possible).

For a complete view of the scalability of our solver, we would like to measure weak scaling. However, it is difficult to find a fair way to measure weak scaling on graphs. We could generate a series of increasingly large random graphs, but in practice, most solvers perform much better on random graphs than real world graphs.

2.4 Conclusions

We have presented a distributed memory graph Laplacian solver for social network graphs. Our solver uses a 2D matrix distribution combined with parallel elimination and unsmoothed aggregation to demonstrate parallel performance on up to 576 processes. Our novel aggregation algorithm can handle arbitrary matrix distributions and forms accurate aggregates while controlling coarse grid complexity on a variety of irregular graphs. The parallel elimination algorithm presented uses generalized matrix products to find elimination candidates independent of matrix distribution. To our knowledge, this is the first distributed memory multigrid solver for graph Laplacians. It enables solving graph Laplacians that would be infeasible to solve on a single computer.

Our solver's robustness is behind that of serial LAMG (as outlined in section 2.3.1) but outperforms the natural parallel extensions of LAMG (no iterate recombination, Jacobi smoothing, and cycle index 1). Further improvement is a topic for future research. Our solver performs well on social network graphs, but WDA is sometimes high for more regular graphs such as road networks. Many such graphs also admit a vertex partition with low edge cut, in which case our 2D distribution may be unnecessary. Extending our solver to handle scaled graph Laplacians (used by some applications) and unsymmetric graphs are other areas for future research.

Chapter 3

Synthetic Bundle Adjustment Problem Creation

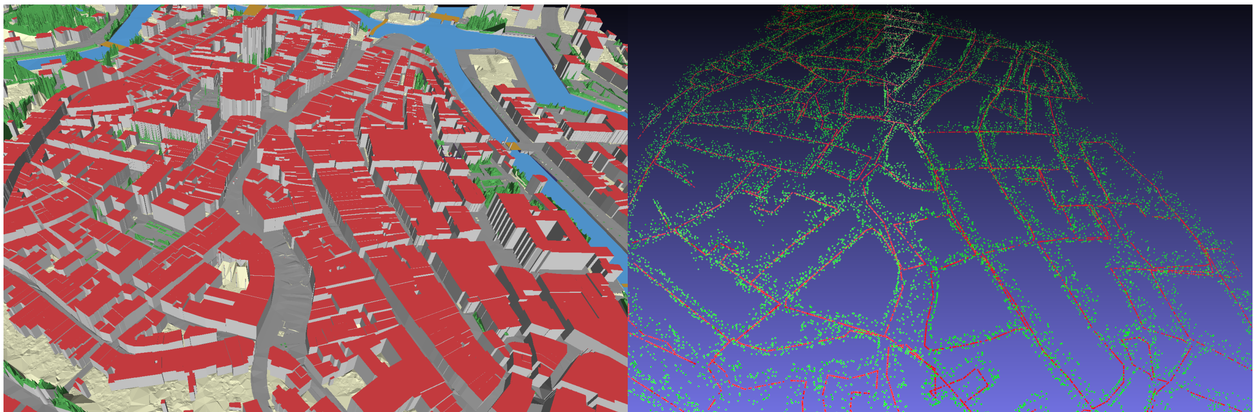


Figure 3.1: Left: 3D model of Zwolle (Netherlands). Right: generated synthetic bundle adjustment dataset.

To test linear solvers for bundle adjustment (chapter 4), we need datasets to test against. Only a couple of real-world datasets are publicly available, namely the Bundle Adjustment in the Large datasets[1] [5] and the 1DSFM datasets[2] [73]. The largest of these datasets contains 15 thousand cameras. As we are interested in evaluating scaling of our algorithms, we require much larger datasets. Also, most of these datasets are "community photo" style, i.e. there are many pictures of the same object. Furthermore, all of these datasets

[1] http://grail.cs.washington.edu/projects/bal

[2] http://www.cs.cornell.edu/projects/1dsfm

contain too many outliers: long range effects are not exposed to the linear solver. We would like datasets with more varied camera counts and visibility structure similar to what we would expect from streetview, so we generate a series of synthetic datasets with these properties.

There are a couple ways we could create our own datasets. We could take existing image datasets, apply a full SfM pipeline, and then apply our bundle adjuster. The resulting datasets should be similar to real world datasets, but running a full SfM pipeline takes a long time. The visibility structure of these datasets is also limited by the structure of the images we use. It is also hard to tell how accurate the bundle adjuster is as there might not be ground truth information for the images. An alternative approach is to generate a synthetic dataset. Synthetic datasets have the benefit of being quick to generate and easy to obtain ground truth info for. They also make it easy to generate a set of similar datasets of different sizes in order to test scalability. Using synthetic datasets also allows us to compare different kinds of noise to see where our algorithm succeeds and falls short.

We generate a ground truth (zero error) bundle adjustment dataset by taking an existing 3D model of a city and drawing potential camera paths through it. We generate random camera positions on these paths, then generate random points on the geometry and test visibility from every camera to every point. We can control the number of cameras and the number of points to generate datasets of different sizes. By choosing different 3D models or different camera paths, we can change the visibility graph between cameras and points. For our datasets, we use a 3D model of Zwolle in the Netherlands[3]. Figure 3.1 shows the 3D model and one of the more complicated datasets we generated.

These datasets do not contain any error, so we add noise into each. The straight forward approach of adding Gaussian noise directly to the camera and point parameters results in a synthetic problem that is easier to solve than the real world problems as it contains no non-local effects. Instead, we add noise similar to what exists in real world bundle adjustment problems:

(1) Camera Drift: initial camera estimates tend to become less accurate the farther they are away from

[3] https://3d.bk.tudelft.nl/opendata/3dfier/

the origin. We model camera drift as

$$d = || - R^T t||, \qquad \text{Distance from origin} \tag{3.1}$$

$$t_{\text{drifted}} = t + \alpha \sigma_1 d^2, \tag{3.2}$$

$$R_{\text{drifted}} = \begin{bmatrix} \sin(\beta \sigma_2 d^{1.2}) & 0 & 0 \\ 0 & \cos(\beta \sigma_2 d^{1.2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} R, \tag{3.3}$$

where R and t and the pose of camera, α and β control the magnitude of the drift, and σ_1 and σ_2 are normal random variables with user specified variance. The translational noise is increased as the square of the distance from the origin and the rotational noise uses the distance to the 1.2 power. These choices are arbitrary.

(2) Imperfect Camera Model: the camera model we use has two parameters for radial distortion. In practice, real world camera lenses exhibit more complex distortion.

(3) Incorrect correspondences: the feature matching process is often imperfect and creates correspondences between two features that are not the same. These are essentially outliers; they will never be aligned and should be discarded. Outliers are usually filtered either by running a nonlinear optimizer multiple times and discarding high-cost points, or by using a robust loss function (see section 4.4.5).

We find that adding camera drift is effective for creating difficult synthetic problems. Adding a little noise to the camera rotational parameters also helps as rotational error is highly nonlinear. We are careful to not add too much noise or too many incorrect correspondences as this leads to problems with many outliers (see section 4.4.5 for a solution).

Code implementing the above synthetic problem generation has been submitted to the Journal of Open Source Software.

Chapter 4

Multigrid for Bundle Adjustment

This chapter is an edited version of work submitted to ECCV '20 done in collaboration with Jed Brown. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0016140. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Performance of a bundler adjuster is bounded by the performance of the linear system solver it uses. The Levenberg-Marquart algorithm that is typically used depends on linear solves to the steps it takes towards the optimum (see section 1.2.2). Existing solutions are slow when problem sizes become large (tens of thousands of cameras) and when long range errors are present. To combat these issues, we develop a new multigrid solver targeted at bundle adjustment. This solver accurately handles long range error and scales better than existing solvers on large, difficult problems.

4.1 Solving the Linear System

Solving the linear system, $J^T J + D$, is the slowest part of Levenberg-Marquardt. This linear system exhibits a special structure: it is composed of four blocks where the diagonal blocks are block diagonal and the off diagonal blocks have arbitrary structure.

$$F = \text{Jacobian restricted to camera parameters}, \tag{4.1}$$

$$E = \text{Jacobian restricted to point parameters}, \tag{4.2}$$

$$J = \begin{bmatrix} F & E \end{bmatrix}, \tag{4.3}$$

$$J^T J + D = \begin{bmatrix} A = F^T F + D_F & F^T E \\ E^T F & C = E^T E + D_E \end{bmatrix}. \tag{4.4}$$

C is a block diagonal matrix with blocks of size 3×3 corresponding to point parameters. A is a block diagonal matrix with blocks of size 9×9 corresponding to camera parameters. $E^T F$ is a block matrix with blocks of size 9×3 corresponding to interaction between points and cameras. A usual trick to apply is using the Schur complement to eliminate the point parameter block C:

$$S = A - F^T E C^{-1} E^T F.$$

S is a block matrix with blocks of size 9×9. C is chosen over A because the number of points is often orders of magnitude larger than the number of cameras. Thus, applying the Schur complement greatly reduces the size of the linear system being solved.

The Schur complement system has structure determined by the covisibility of cameras: if c_i and c_j both observe the same point, then the block S_{ij} is nonzero. In practical terms, S tends to be dense when all the images are of the same object, for example, tourist photos of the Eiffel Tower. S tends to be sparse when the images cover a large area, like images taken from a car as it drives around a city (we call this situation *street view*). We will focus on solving sparse S for this paper.

This linear system is normally not solved to a tight tolerance. Usually, a fairly inexact solve of the linear problem can still lead to good convergence in the nonlinear problem [5]. As the nonlinear problem gets closer to a minima, the accuracy of the linear solve should increase. The method for controlling the linear solve accuracy is called a *forcing sequence*. Ceres Solver [4], the current state of the art nonlinear least-squares solver, uses a criterion proposed by Nash and Sofer [50] to determine when to stop solving the linear problem:

$$Q_n = \frac{1}{2} x^T A x - x^T b, \tag{4.5}$$

$$\text{stop if } i \frac{Q_i - Q_{i-1}}{Q_i} \leq \tau, \tag{4.6}$$

where i is the current conjugate gradients iteration number and τ is the tolerance from the forcing sequence. It is important to note that the occurrence of the iteration number in the criteria means that more powerful preconditioners end up solving the linear problem to a tighter tolerance.

When using a simple projective model, the bundle adjustment problem is ill-conditioned. There are four main causes:

(1) The large difference in scale between rotational and translational parameters (rotations are in the range 0–π, translations in the range 0–1000) causes scale issues in the Jacobian.

(2) The distortion model used is highly nonlinear (it contains a fourth order term).

(3) Projecting points into the camera frame means that moving points that are closer has a much larger effect on the residual than moving father away points the same distance.

(4) Rodrigues vectors have a singularity when their angle of rotation is zero.

There are a number of solutions proposed for these issues. For example, in [57], Qu adaptively reweights the residual functions and uses a local parameterization of the camera pose to improve conditioning. These improvements are largely orthogonal to improving the performance of the linear solver. For this thesis we will use the simple model proposed earlier as it is used in the real world and would like our solver to handle ill-conditioned matrices. We do use a diagonal scaling of the Jacobian as it can be applied to any problem:

$$\mathcal{D} = \text{diag}(J^T J), \tag{4.7}$$

$$\tilde{J} = \mathcal{D}^{-\frac{1}{2}} J. \tag{4.8}$$

This scaling is essential to improve the conditioning enough that matrix-matrix products with J do not suffer from excess roundoff error.

4.2 Related Work

There are a variety of ways to solve S. Konolige uses a sparse direct Cholesky solver [37]. Sparse direct solvers are often a good choice for small problems because of their small constant factors. For large problems with 2D/planar connectivity, sparse direct methods require $O(n^{1.5})$ time and $O(n \log n)$ space when small vertex separators exist (a set of vertices whose removal splits the graph in half) [44]. In street view problems, camera view overlap in street intersections creates large vertex separators, making sparse direct solvers a poor choice for large problems. To improve scaling, Agarwal et al. propose using conjugate gradients with Jacobi preconditioning [5]. Kushal and Agarwal later extend this work with block-Jacobi and block-tridiagonal preconditioners formed using the *visibility*, or number of observed points in common between cameras [40]. Jian et al. propose using a preconditioner based off of a subgraph of the unreduced problem similar to a low-stretch spanning tree [33].

4.3 Multigrid for Bundle Adjustment

4.3.1 Nullspace

Fast convergence of multigrid requires satisfaction of the *strong approximation property*:

$$\min_u ||e - Pu||_A^2 \leq \frac{\omega}{||A||} \langle Ae, Ae \rangle,$$

for any fine-grid error e and constant ω determining convergence rate [47, 59, 67]. To satisfy this condition, es for which $||Ae||$ is small (near-nullspace vectors) must be accurately captured by P. For our bundle adjustment formulation, $J^T J$ has a nullspace, N, with 7 degrees of freedom corresponding to the free modes of the nonlinear problem. Specifically, the nonlinear problem can be freely translated (in 3 directions), rotated (in 3 directions), and scaled. We compute these vectors analytically:

$$N_t^1 = -R \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T, \quad (4.9)$$

$$N_t^2 = -R \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T, \quad (4.10)$$

$$N_t^3 = -R \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T, \quad (4.11)$$

$$N_t^4 = t, \tag{4.12}$$

$$\begin{aligned} N_R^{5,6,7} =& \frac{1}{2} \left|\csc\left(\frac{a}{2}\right)\right| \left(r \left(-a \cos\left(\frac{a}{2}\right) + \right.\right. \\ & \left. \sqrt{2 - 2\cos(a)} \right) (-r \cdot a) + \\ & \left.\left. a \left(-a \cos\left(\frac{a}{2}\right) + (r \times -a) \sin\left(\frac{a}{2}\right)\right)\right)\right), \\ \text{where } a =& \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T, \\ & \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T, \\ & \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T. \end{aligned} \tag{4.13}$$

N^1 to N^3 are the linearized translations, N^4 is the linearized scaling, and N^5 to N^7 are the linearized rotations. We empirically verify that N is in the nullspace of S. When the damping matrix D is small, this nullspace is near-nullspace vectors, K, of $J^T J + D$. For the Schur complement system, the near-nullspace of S is the near-nullspace of $J^T J + D$ restricted to the camera parameters. We augment the near-nullspace with 9 columns that are constant on each of the 9 camera parameters.

4.3.2 Aggregation

Algorithm 5 Aggregation

```
 1: function AGGREGATE(strength-of-connection matrix G of size n × n)
 2:     for i ∈ 1..n do
 3:         if i is unaggregated then
 4:             for j ∈ G_{i,:} sorted by largest to smallest do
 5:                 if j is unaggregated then
 6:                     form new aggregate with i and j
 7:                     break
 8:                 else if j is in aggregate k and size(k) < 20 then
 9:                     add i to aggregate k
10:                     break
11:                 end if
12:             end for
13:         end if
14:     end for
15: end function
```

The multigrid aggregation algorithm determines both how quickly the solver converges and the time

it takes to apply the preconditioner. Choosing aggregates that are too large results in a cheap cycle that converges slowly. On the other hand, if aggregates are too small, the solver will converge quickly but each iteration will be computationally slow. The aggregation routine needs to strike the right balance between too large and too small aggregates.

Typical aggregation routines for multigrid form fixed diameter aggregates by clustering together a given “root” node with all its neighbors. This technique works well on PDE problems where the connectivity is predictable and each degree of freedom has low degree. Bundle Adjustment does not necessarily have these characteristics. Street view-like problems might have low degree for road sections that do not overlap, but when roads intersect, some dof’s can be connected to many others. Choosing one of these well connected dof’s as the root of an aggregate creates a too aggressive coarsening.

Aggregation routines for non-mesh problems exist, for example, for graph Laplacians [38, 46]. These routines have to contend with dofs that are connected to a majority of other dofs; something we do not expect to see in street view-like datasets. Instead, we use a greedy algorithm that attempts to form aggregates by aggregating unaggregated vertices with their “closest” connected neighbor and constrains the maximum size of aggregates to prevent too aggressive coarsening.

Closeness of dofs is determined by the *strength-of-connection* matrix. Almost all multigrid aggregation algorithms use this matrix as an input to determine which vertices should be aggregated together. The strength-of-connection matrix can be created based only using matrix entries (for example, the affinity [46] and algebraic distance metrics [19]) or use some other, geometric information. For bundle adjustment, this other information can be camera and point positions or the visibility information between them. The strength-of-connection metric we choose to use is the visibility metric used by Kushal and Agarwal in [40]. We tried other metrics, like including the percentage of image overlap between two cameras, but the visibility metric remained superior. The visibility strength-of-connection matrix, G, is defined as:

$$G_{i,j} = \begin{cases} 0, & i == j, \\ \frac{v_i^T v_j}{||v_i|| ||v_j||}, & \text{otherwise}, \end{cases} \tag{4.14}$$

$$v_{k_l} = \begin{cases} 1, & \text{camera } k \text{ sees point } l, \\ 0, & \text{otherwise.} \end{cases} \tag{4.15}$$

Most aggregation routines for PDE's enforce some kind of diameter constraint on aggregates. We find that for our problems this is not necessary. However, we force aggregates to contain no more than 20 dofs, to ensure our aggregates do not become very large. In practice, we see that aggregate sizes are usually in the range for 8 to 3, with the mean aggregate size usually just a little more than 3.

4.3.3 Prolongation

We use a standard multigrid prolongation construction technique [1, 67]. For each aggregate, the nullspace is restricted to the aggregate, a QR decomposition is applied, and $\mathcal{Q}$ becomes a block of P while $\mathcal{R}$ becomes a block of the coarse nullspace:

$$\mathcal{Q}_{\text{agg}}\mathcal{R}_{\text{agg}} = N_{\text{agg}} \text{ forall agg} \in \text{ aggregates}, \tag{4.16}$$

$$P = \Pi \begin{bmatrix} \mathcal{Q}_1 & & \\ & \ddots & \\ & & \mathcal{Q}_n \end{bmatrix}, \tag{4.17}$$

$$N_{\text{coarse}} = \Pi \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix}. \tag{4.18}$$

Here Π is a permutation matrix from contiguous aggregates to the original ordering. Using the QR decomposition frees us from having to compute the local nullspace and represent it on the coarse level (this would require computing the center of mass of each aggregate). Our near-nullspace has dimension 16 (7 from $J^T J$'s nullspace, 9 from per dof constant vectors), so each of our coarse level matrices has 16 by 16 blocks.

4.3.4 Smoother

We use a Chebyshev smoother [3] with a point block Jacobi matrix. We find the Chebyshev smoother to be more effective than block-Jacobi and Gauss-Seidel smoothing. The Chebyshev smoother does come

with a disadvantage: it requires an estimate of the largest eigenvalue, λ_{max}, of $D^{-1}A$. Like Tamstorf et al., we find that applying generalized Lanczos on $Ax = \lambda Dx$ is the most effective way to find the largest eigenvalue [67]. This eigenvalue estimate is expensive, so we limit it to 5 applications of the operator. We use $1.1\lambda_{\text{max}}$ for the high end of the Chebyshev bound and $0.3\lambda_{\text{max}}$ as the lower end. However, the superior performance of the Chebyshev smoother outweighs its increased setup cost. We also tried using the Gershgorin estimate of the largest eigenvalue, but that proved to be very inaccurate (by multiple orders of magnitude). We apply two iterations of pre-smoothing and two of post-smoothing.

4.3.5 To Smooth or Not To Smooth

Aggregation-based multigrid uses prolongation smoothing in order to improve convergence [72]. Smoothing the prolongation operator is sufficient to satisfy the strong approximation property and achieve constant iteration count regardless of problem size. Usually, smoothing the prolongation operator improves convergence rate at the cost of increased complexity of the coarse grids. In PDEs and other problems with very regular connectivity, this trade off is worthwhile. However, in other problems, like in irregular graph Laplacians, irregular problem structure causes massive fill-in—coarse grids become dense [46]. The street view-based bundle adjustment problems we are working with appear to be similar in structure to PDE based problems: the number of nonzeros per row is bounded and the diameter of the problem is relatively large. However, when we apply prolongation smoothing to our multigrid preconditioner, we see large fill-in in the coarse grids, similar to what happens in irregular graph Laplacians. Although the nonzero structure of street view bundle adjustment appears similar to PDEs, it still has places where dofs are coupled with many other dofs—places where large fill-in occurs. These places could be landmarks that are visible from far away or intersections where there is a large amount of camera overlap. The large fill-in on the coarse grid makes the setup phase too expensive to justify the improved performance in the solve phase (see figure 4.7). Choosing not to smooth aggregates means our solver does not scale linearly, but it does scale better than any of the current state of the art solvers.

4.3.6 V-Cycles vs W-Cycles

Two common areas where a given multigrid implementation can struggle are the coarse grid solve accuracy and the fine grid smoother. To test if our coarse grid solve is accurate, we compared performance of our multigrid preconditioner using W-cycles vs V-cycles. W-cycles solve the coarse grid multiple times for a more accurate solution. If W-cycles perform better, then we know our coarse grid solve is not accurate enough.

Figure 4.1 shows the result of this experiment. Each plot measures the nonlinear objective function value vs cumulative solve time on a number of different problems. In every problem, the solve time with V-cycles is approximately equal to the solve time with W-cycles. From this we conclude that the coarse grid solve is accurate enough.

4.3.7 Implicit Operator

On many bundle adjustment problems, it is often faster to apply the Schur complement in an implicit manner, rather than constructing S explicitly [5]. That is, we can apply manifest Sx for a vector x as $Ax - F^T(E(C^{-1}(E^T(Fx))))$. As the conjugate gradients algorithm (CG) requires only matrix-vector products, we can use the implicit matrix product with it for improved performance. An issue arises when we use a preconditioner with CG: the preconditioner often needs the explicit representation of S. For block-Jacobi preconditioning, Agarwal et al. [5] construct on the relevant blocks of S. The same technique is used by Kushal and Agarwal in their visibility-based preconditioner [40]. For Algebraic multigrid, the explicit matrix representation is needed to form the Galerkin projection P^TSP. We could use the implicit representation with the Galerkin projection, $P^T(A(Px)) - P^T(F^T(E(C^{-1}(E^T(F(Px))))))$, but then we are paying the cost of the full implicit matrix at each level in our hierarchy. We instead compute the cost of using the implicit vs explicit product on each level of our hierarchy and choose the cheapest one. In our tests we create both the implicit and explicit representations for each level, but only use the most efficient one. It would be possible to create only the needed representation on each level, but we have not explored the costs. This may actually have a large performance impact as the Galerkin projection requires

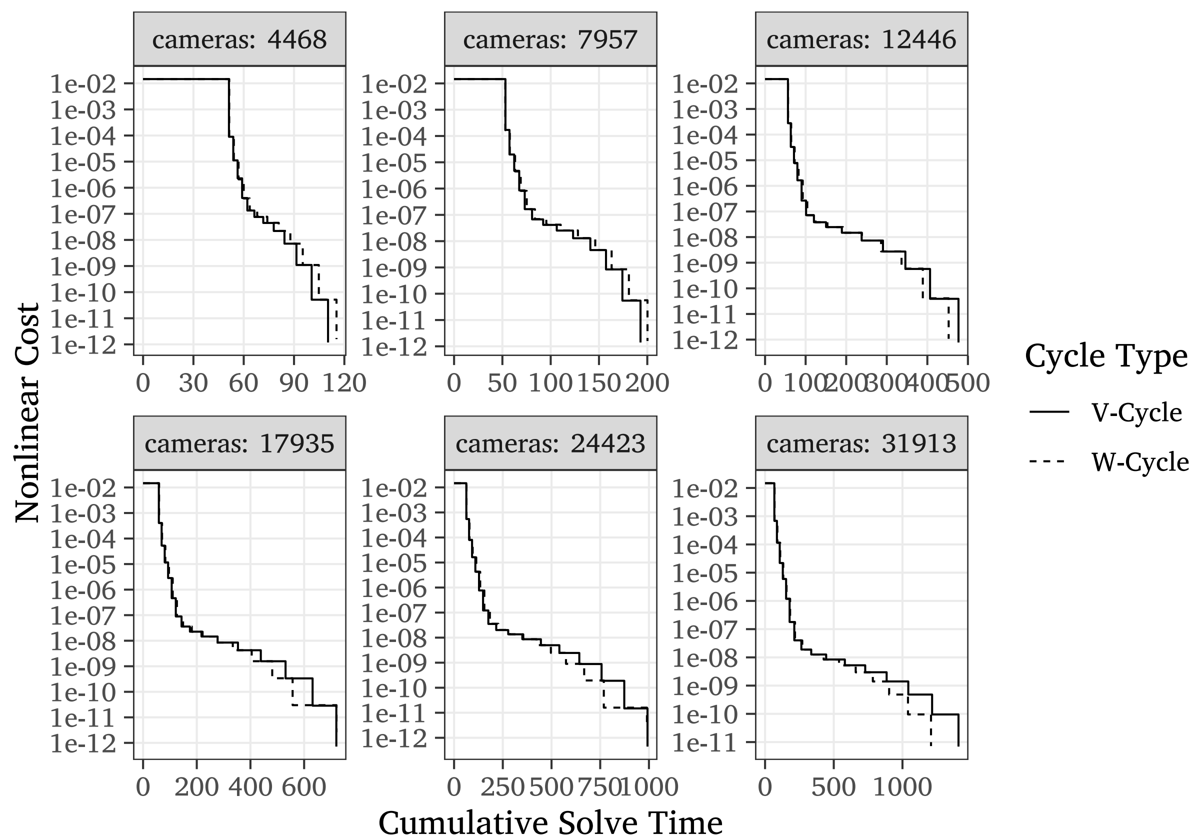


Figure 4.1: Nonlinear objective function values vs cumulative solve times for multigrid with V-cycles and W-cycles on a set of increasingly larger synthetic problems. Multigrid performs about the same with V-cycles and W-cycles, indicating that coarse grid solves are accurate enough with V-cycles.

an expensive matrix-triple product (currently the most expensive part of setup).

This choice of implicit vs explicit operator highlights an important point of using multigrid for bundle adjustment: the explicit operator might not be computationally feasible. In situations where the visibility graph becomes dense, the explicit matrix will grow as $O(n^2)$. Like the current state of the art (visibility based preconditioning), we ignore these situations and instead focus on problems where the visibility structure is sparse. It may be possible to use some kind of sparsification before performing the linear solve in order to avoid this problem.

4.4 Results

We tested our multigrid preconditioner against point block Jacobi and visibility-based block Jacobi preconditioners on a number of synthetic problems (we found the visibility-based tridiagonal preconditioner to perform similarly to visibility-based block Jacobi, so we omit it). Our test machine is an Intel Core i5-3570K running at 3.40GHz with 16GB of dual-channel 1600MHz DDR3 memory. For large problems, we use NERSC's Cori—an Intel Xeon E5-2698 v3 2.3 GHz Haswell processor with 128 GB DDR4 2133 MHz memory. We use Ceres Solver [4] to perform our nonlinear optimization as well as for the conjugate gradient linear solver. Ceres Solver also provides the point block Jacobi and visibility based preconditioners. We terminate the nonlinear optimization at 100 iterations or if any of Ceres Solver's default termination criteria are hit. Our initial trust region radius is 1e4. We use a constant forcing sequence with tolerance τ. Results are post processed to ensure that all nonlinear solves for a given problem end at the same objective function value. For some preconditioners (like our multigrid), this significantly impacted the total number of nonlinear iterations taken (see section 4.4.1).

Our solver is written in Julia [13] and uses SuiteSparse [24] for its Galerkin products. Our implementation is not heavily optimized (unlike the point block Jacobi and visibility based preconditioners). We do not cache the sparse matrix structure between nonlinear iterations and reallocate almost all matrix products. Furthermore, the Julia code allocates more and is slower than it could be if written in C or C++. Jacobian matrices are copied between Julia and Ceres Solver, leading to a larger memory overhead. We do

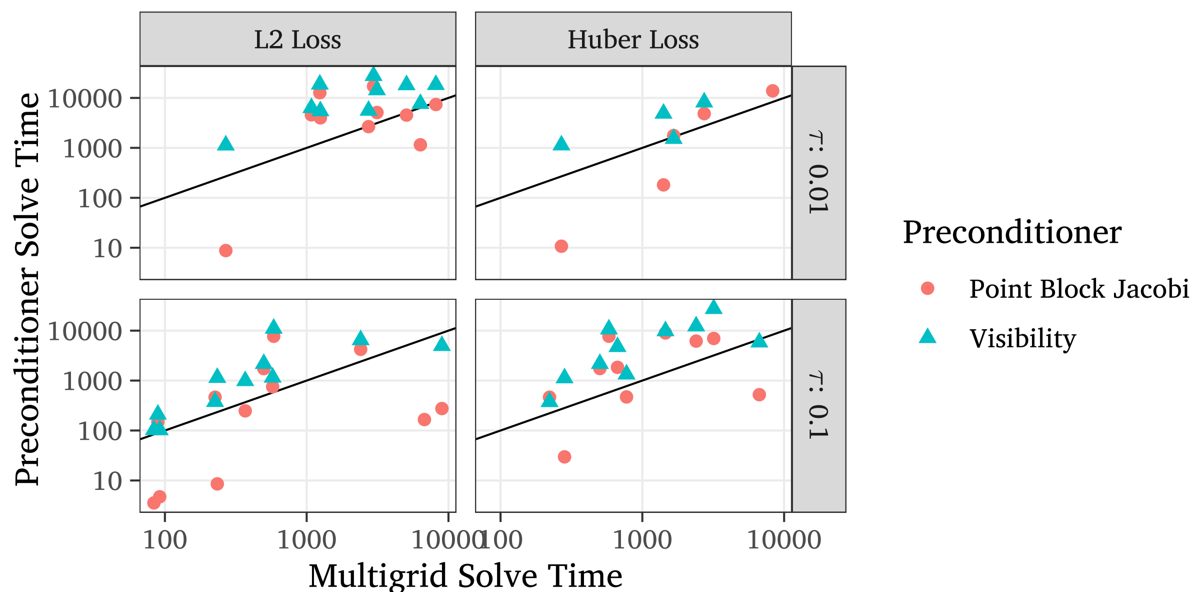


Figure 4.2: Preconditioner solve time versus multigrid solve time for a set of synthetic problems with varying number of cameras, visibility structure, and noise. Solve time is measured as total time spent in the linear solver (setup and solve) for all nonlinear iterations to a certain problem dependent tolerance. Points above the diagonal (black line) indicate the problem was solved quicker with multigrid than the given preconditioner, points below indicate that multigrid was slower. Vertical columns of plots have use the same loss function. Horizontal plot rows have the same linear solve tolerance τ. For the majority of cases, multigrid performs better than all the other solvers.

not use a sparse matrix format that exploits the block structure of our matrices or use matrix-multiples that exploit this structure. All of this is to say that our method could be optimized further for potentially greater speedup.

Still, our multigrid preconditioner performs better than point block Jacobi and visibility based preconditioners on most large problems. Figure 4.2 shows the relative solve time of other preconditioners vs our multigrid preconditioner for a variety of synthetic problems. Our preconditioner is up to 13 times faster than point block Jacobi, and 18 times faster than visibility based preconditioners. Median speedup is 1.7 times faster than point block Jacobi, and 2.8 times faster than visibility based preconditioners. This includes cases where problems are large, but not difficult; a situation where our solver performs poorly. On smaller problems (with less than 1000 cameras), our solver is significantly slower than direct methods.

On problems where the geometry is simpler, point block Jacobi normally outperforms visibility based preconditioners and our solver. This is because the linear problems are relatively easier to solve and the visibility based methods cannot recoup their expensive setup cost. On more complicated problems (when the camera path crosses itself), the difficulty of the solve makes the high setup cost of the visibility based methods worthwhile. We find that these more complicated problems are also where our multigrid preconditioner has a larger speedup over the other methods. We believe that this is because the multigrid preconditioner does a good job of capturing long range effects in the problem.

4.4.1 Solver Accuracy

Our multigrid preconditioner is a more accurate preconditioner than point block Jacobi and visibility based methods. In general, preconditioners like point block Jacobi converge fast in the residual norm, but converge slower in the true error norm. Multigrid tends to converge similarly in the error norm and the residual norm. Figure 4.5 shows this behavior on a synthetic problem. This behavior is reflected in the nonlinear convergence when using our solver vs point block Jacobi. Each nonlinear iteration with multigrid reduces the objective function by a larger value than point block Jacobi, indicating that the multigrid solution was more accurate. See figure 4.3 for plots of this behavior on some of our datasets. Also interesting to note in this figure is the slope of convergence. In almost all the plots, the solvers first

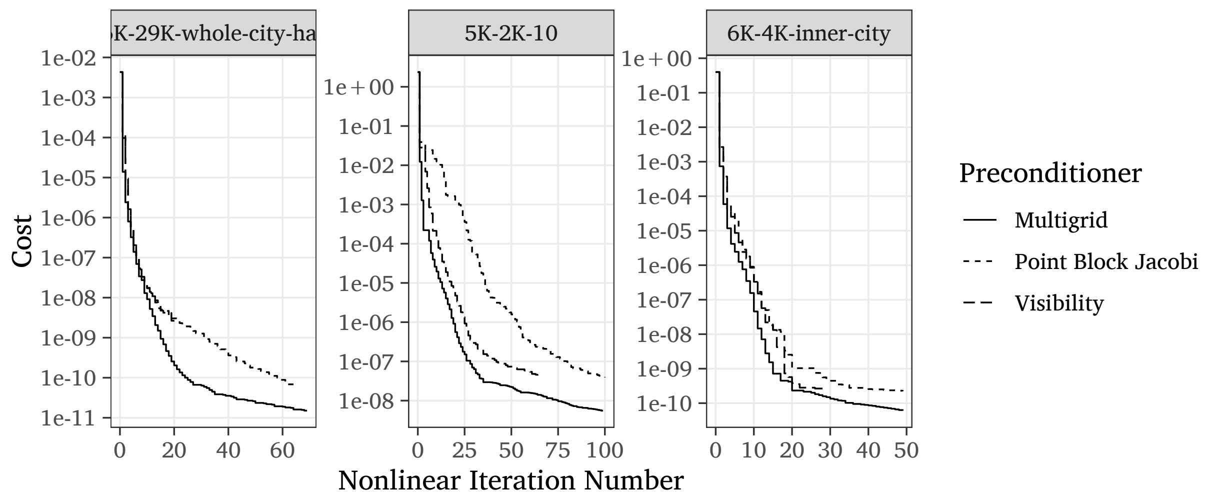


Figure 4.3: Objective function value vs nonlinear iteration number for a variety of synthetic problems with varying problem structure. Our multigrid solver tends to reach that value in fewer iterations than the other solvers because it is more accurate for a given solve tolerance. For solves where a high accuracy is required, or where Jacobian calculation is expensive, our solver is a good choice.

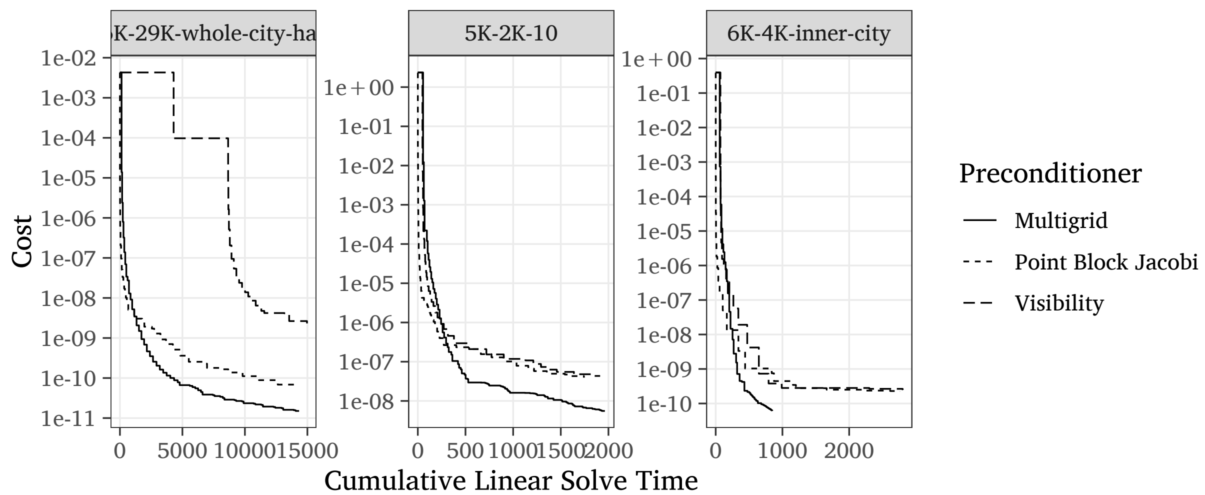


Figure 4.4: Objective function value vs cumulative time for a variety of synthetic problems with varying problem structure. Although multigrid is slow initially, its handling of long range error means it converges quickly for longer than point block Jacobi and visibility based preconditioning. Note that visibility based preconditioning is very slow in the first couple of iterations. We believe this is a scalability bug in its setup phase.

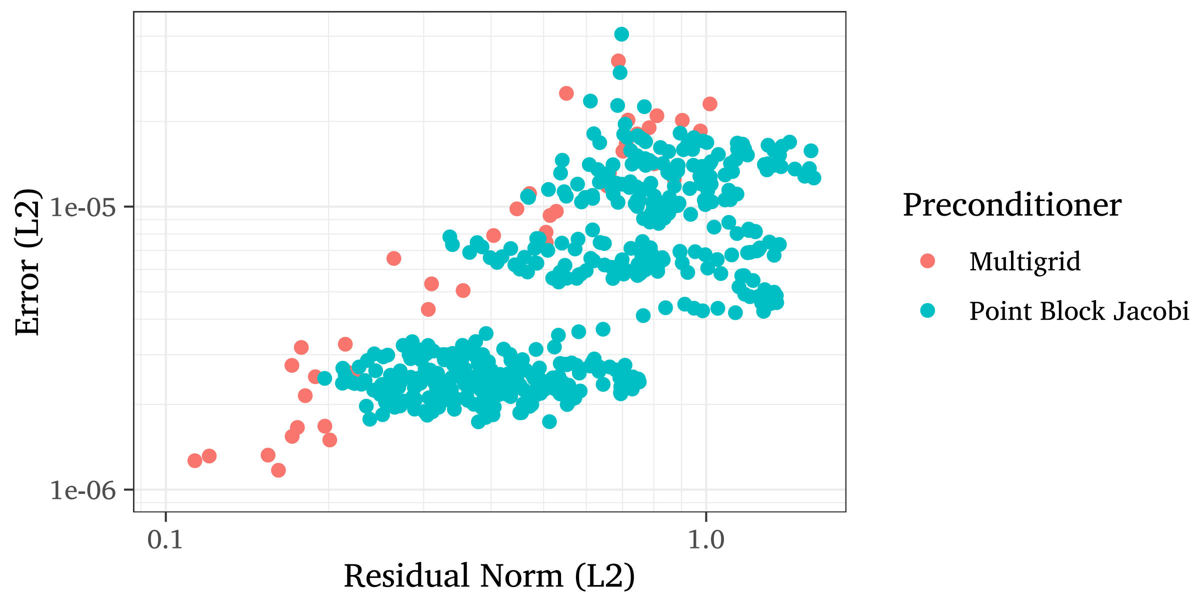


Figure 4.5: Residual norm vs error norm on a synthetic problem with long range noise. Note that for a given error, multigrid has a higher residual norm than point block Jacobi.

converge quickly then hit a point where they start converging more slowly. Our multigrid solver also follows this characteristic, but converges more steeply in the first phase and continues converging quickly for longer. We believe this is because our solver more accurately captures long range effects. For nonlinear optimization problems where a high degree of accuracy is required, this behavior makes our multigrid preconditioner even more performant than existing solvers.

For solves where τ is smaller (0.01), our solver performs better than point block Jacobi. When τ is larger, our solver is generally slower than point block Jacobi because the setup cost of our solver is not amortized. In general, our solver is a good choice when when tight (small) solve tolerances are used or when the linear problems are hard to solve.

4.4.2 Scaling

For larger problem sizes, the algorithmic complexity of different solution techniques begins to dominate over constant factors. It is well known that solving a second order elliptic system (such as elasticity) on a $\sqrt{n} \times \sqrt{n}$ grid using conjugate gradients with point block Jacobi preconditioning requires $O(n^{1/2})$ iterations, for a total cost of $O(n^{1.5})$ [68]. We expect to interpret the global coupling and scaling of bundle adjustment similarly, in terms of diameter of the visibility graph, which has 2D grid structure for street view data in cities. If the structure is not two dimensional, say for a long country road, then we would expect the bound to be $O(n^2)$. We expect that visibility based methods also scale as $O(n^{1.5})$, but with different constant factors as they cannot handle long range effects. Multigrid can be bounded by $O(n)$, but this requires certain conditions on the prolongation operator that we do not satisfy (specifically, not smoothing the prolongation operator means that we do not satisfy the strong approximation property). Empirically, we find that our multigrid technique does not scale linearly with problem size, but still scales better than other preconditioners (figure 4.6).

To empirically verify the scaling of visibility-based methods and our multigrid method, we construct a series of city block-like problems with increasing numbers of blocks. Increasing the number of city blocks instead of adding more cameras to the same structure means that the test problems have increasing diameter. We add noise that looks like a sine wave to the problem to induce long range errors. Figure 4.6

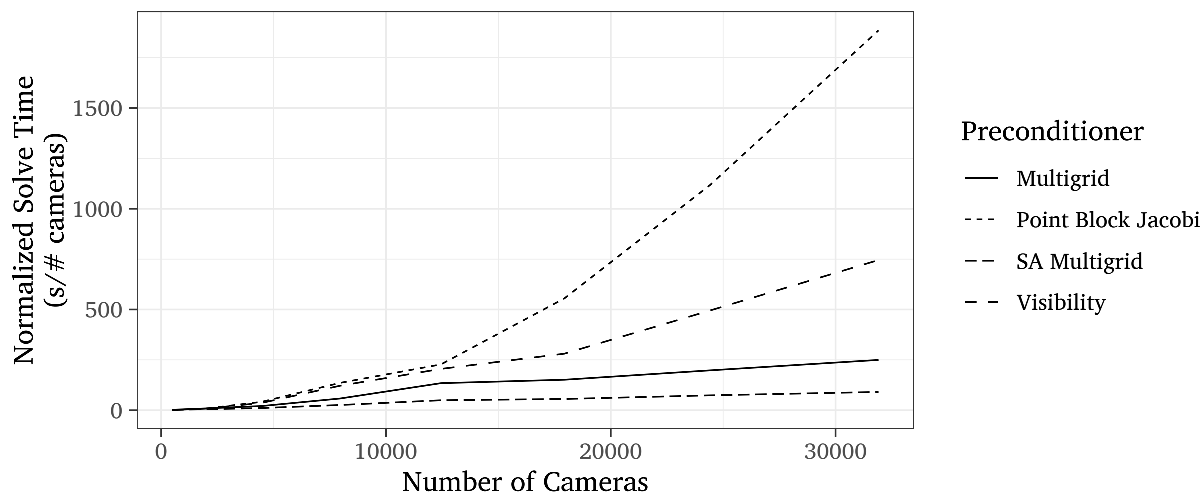


Figure 4.6: Linear solver scaling experiment on a series of increasingly larger grids with long range noise only. Grid size is on the order of $\sqrt{\text{number of cameras}} \times \sqrt{\text{number of cameras}}$. The y-axis is a measure of linear solver solve time (not including linear solver setup) per camera. A horizontal trend indicate that a solver is scaling linearly with the number of cameras. Slopes greater than zero indicates the solver is scaling superlinearly. We see the expected behavior that Multigrid scales close to linearly while visibility and point block Jacobi scale superlinearly. Smoothed aggregation multigrid has the best scaling, but its setup phase is prohibitively expensive.

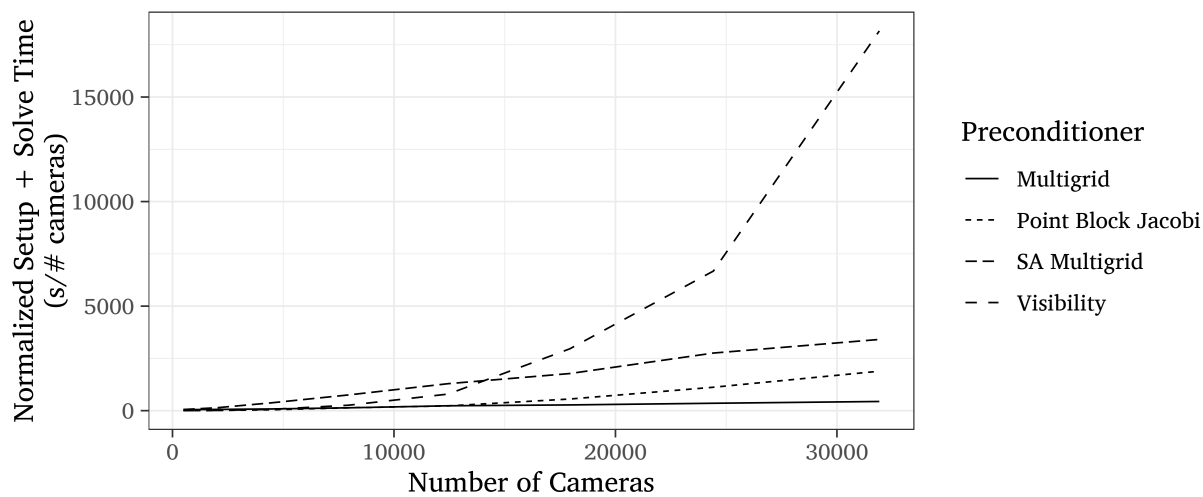


Figure 4.7: Linear solver scaling experiment on a series of increasingly larger grids with long range noise only. The setup is the same as figure 4.6, except this plot contains the setup and solve times of the linear solver. Our multigrid preconditioner with smoothed aggregation performs worse than point block Jacobi and our multigrid preconditioner without smoothing due to the high setup cost of prolongations smoothing. The visibility preconditioner appears to be scaling as $O(n^3)$. Comparing to the plot without setup time, we see that this poor scaling appears entirely in the setup phase.

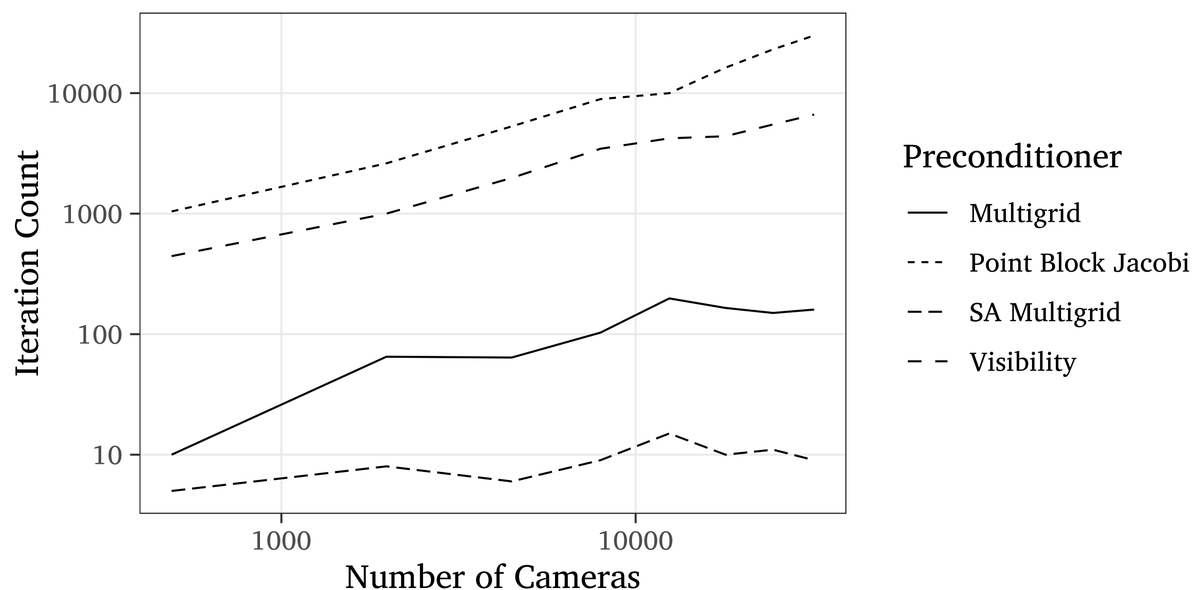


Figure 4.8: Linear solver scaling experiment on a series of increasingly larger grids with long range noise only. The setup is the same as figure 4.6, except this plot measures the iteration count of the solve vs the number of cameras. We would expect the smoothed aggregation multigrid to have zero slope as it should be an $O(n)$ solver. The noise introduced into the problem could contribute to the superlinear behavior we are seeing.

shows the results of this experiment. Surprisingly, point block Jacobi is scaling as $O(n^2)$, which indicates that bundle adjustment is more similar to a shell problem than an elasticity problem. Still, our solver is faster than other solvers in both solve time and setup plus solve time (figure 4.7) for large problems.

4.4.3 Eigenvalues

As mentioned in the previous section, we would expect the difficulty of the linear problem to grow with the condition number and the diameter of the problem. However, when we compute the eigenvalues, we find that the condition number of the problem does not grow with the diameter of the problem. Figure 4.11 plots condition number vs diameter for for a set of increasing larger synthetic city block problems (the same as in section 4.4.2). However, we would expect the iteration counts to grow as the square root of the condition number, but this is not the case in figure 4.11. The cause of this discrepancy is unclear.

4.4.4 Parallelism

Our solver is currently single threaded. Most of the time spent in the solver is in the linear algebra routines, so using a parallel linear algebra framework is an easy way to parallelism. The only non-linear algebra part of our multigrid solver is aggregation. There are parallel aggregation techniques, but they are not suited for the irregular structure present in bundle adjustment problems.

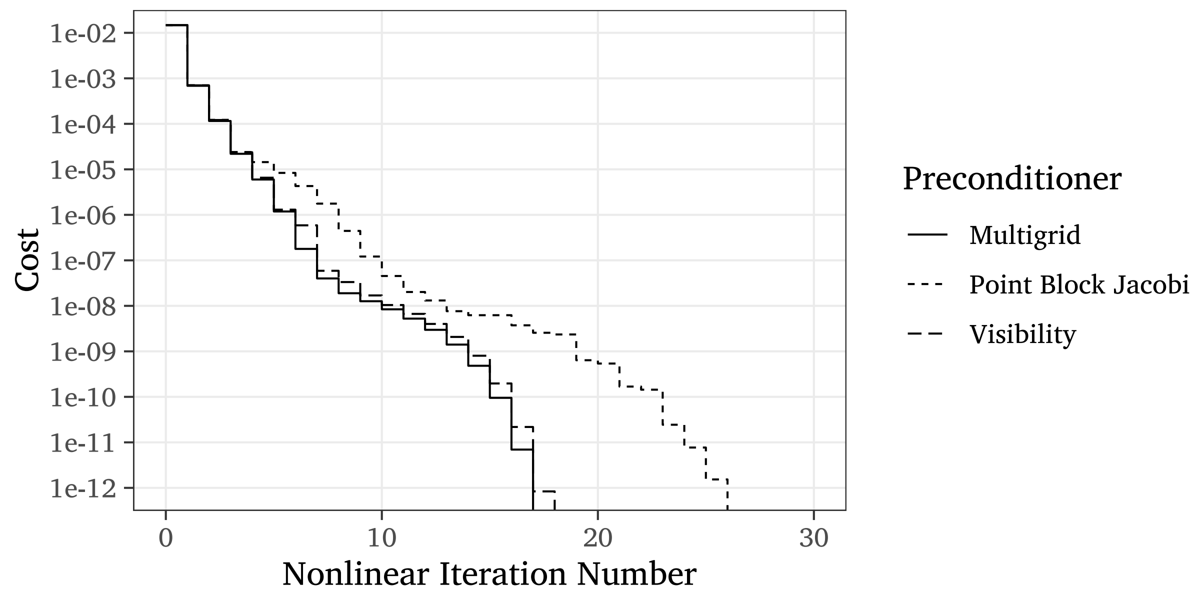


Figure 4.9: Nonlinear cost vs nonlinear iteration number on a 40 by 40 synthetic city grid. The linear solve tolerance is $\tau = 0.01$ with a Huber loss function to expose long range error. The cost for solves with point block Jacobi lags behind solves with multigrid or visibility preconditioners because point block Jacobi is a less accurate solver.

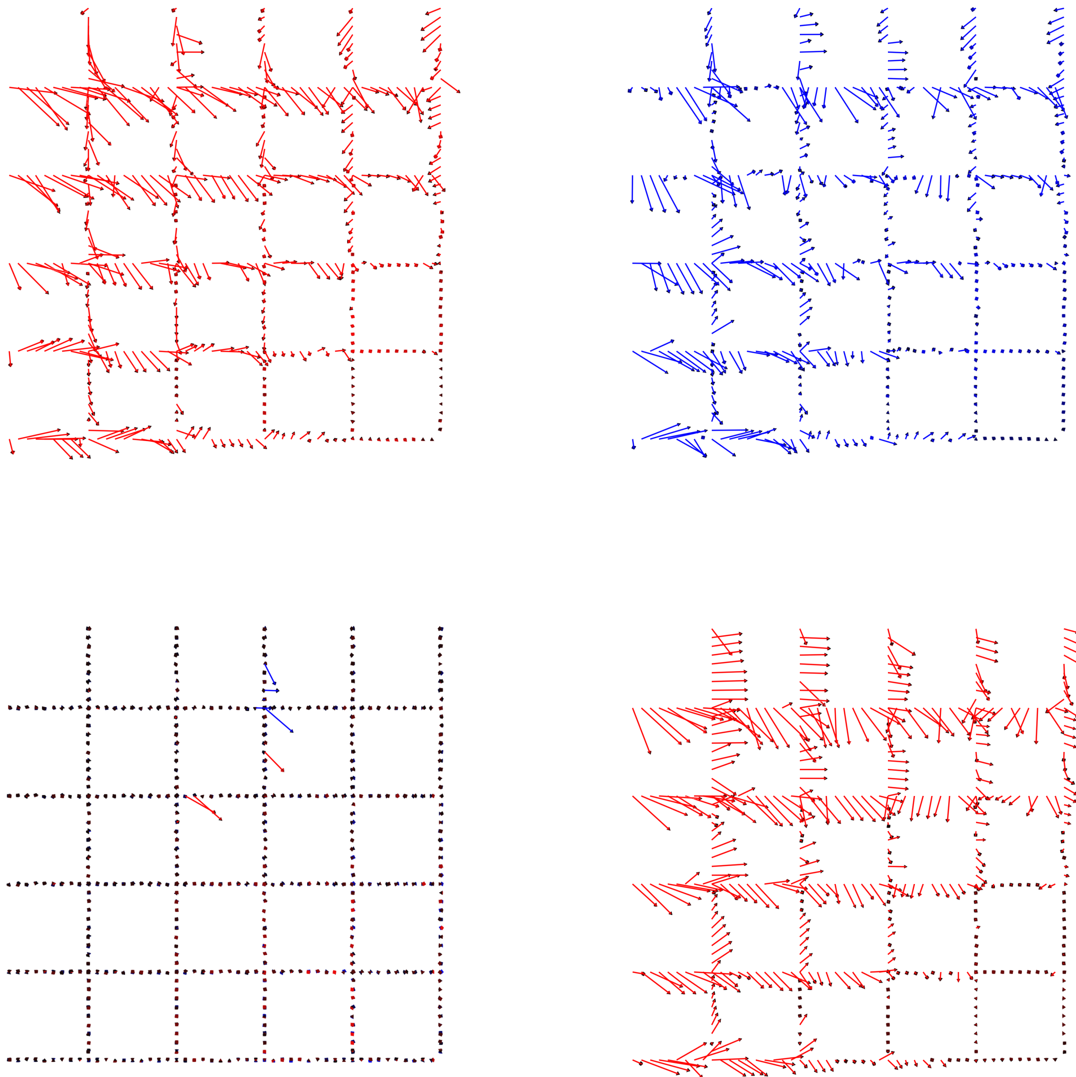


Figure 4.10: Eigenvector plots for a synthetic problem. In clockwise order from top left: 1. Smallest eigenvector on the first nonlinear solve iteration 2. Second smallest eigenvector on the first nonlinear iteration 3. Smallest eigenvector on the 10th nonlinear iteration 4. First and second largest eigenvectors on the first nonlinear iteration. The smallest eigenvectors remain the same between the first and 10th nonlinear solve, but their respect eigenvalues are different by a factor three orders of magnitude.

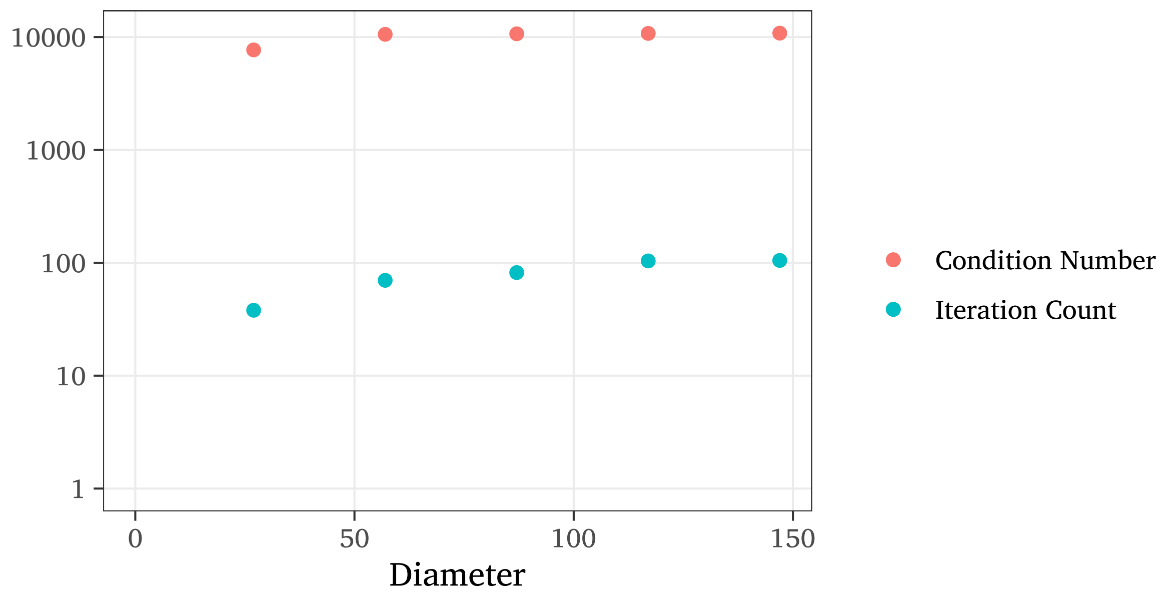


Figure 4.11: Condition number and linear solver iteration count vs problem diameter on a series of increasingly larger synthetic problems. Although the linear problems becoming increasing difficult to solve with larger diameter, the condition number does not reflect this.

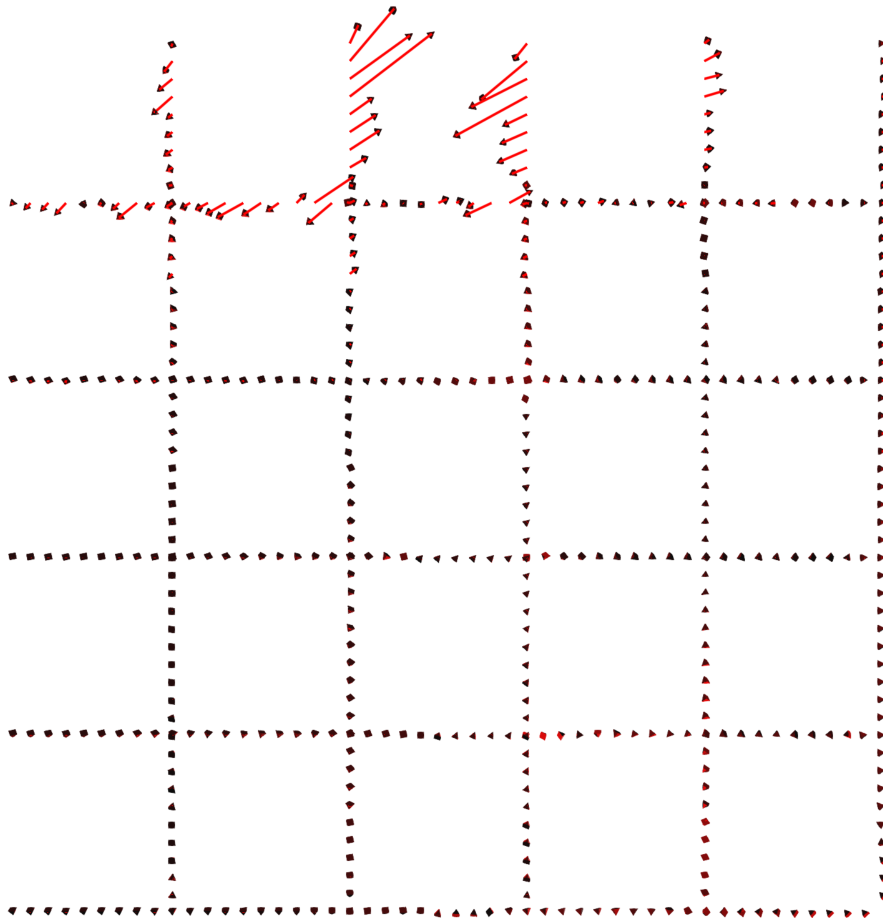


Figure 4.12: Plot of the smallest eigenvector at the nonlinear solution of a synthetic bundle adjustment problem. Largest movements are in the direction that the cameras are facing.

4.4.5 Robust Error Metrics

Often, there are outlying points in bundle adjustment problems. These are the product of incorrect correspondences, points that are too close to accurately track, or points with very poor initialization. In any case, outlying points make up a disproportionate amount of the nonlinear cost function (due to the quadratic scaling of the cost). Levenberg-Marquardt attempts to minimize the residual, and the quickest way it does is to focus on the most extreme outliers. This effectively masks the presence of long-range error. The usual solution is to use a robust loss function. Robust loss functions are quadratic around the origin, but become linear the farther they get from the origin. The robust loss function we use is Huber loss,

$$\text{loss}(x) = \begin{cases} x, & x \leq 1, \\ 2\sqrt{x} - 1, & x > 1, \end{cases}$$

where x is the squared L2 norm of the residuals.

Point block Jacobi is a local preconditioner: it is effective at resolving noise in a small neighborhood. Without a robust loss function point block Jacobi is quick because it "fixes" outliers in a couple iterations. A robust loss function exposes long-range noise making point block Jacobi slow. However, multigrid is more effective at addressing long range error, so it is a comparatively faster solver when used with a robust loss function.

4.4.6 When to Use Multigrid

Our multigrid solver is most effective in addressing long range error. It also scales much better than point block Jacobi and visibility based preconditioners on increasing problem size. However, its expensive setup time needs to be amortized by its improved solve times. With these points in mind, we recommend the following choices of linear solver:

- If your problem is small (less than 1000 cameras), use a direct solver.
- If your problem has many outliers, for example, in the initial iterations of bundle adjustment, use point block Jacobi.

- If your problem is large and contains long range errors, use our multigrid solver.
- If your problem is large and requires high solve tolerances, for example, if Jacobian evaluation is very expensive, use our multigrid solver as its setup phase is amortized by the high cost solve phase.

4.5 Conclusion & Future Work

We present a multigrid preconditioner for conjugate gradients that performs better than any existing solver on bundle adjustment problems with long range effects or problems requiring a high solve tolerance. In tests on a set of large synthetic problems, our solver is up to 13 times faster than the next best solver. Our solver is tailored for a specific kind of bundle adjustment problem: a 9-parameter camera model with reprojection error. Generalizing this solver to different kinds of camera models would require computing a new analytical nullspace. For most models, this should just involve finding the instantaneous derivatives of the 7 free modes (3x translation, 3x rotation, 1x scaling). It would also be possible to use an eigensolver to find the near-nullspace at an increase in setup time cost [17].

In future work we would like to find a way to automatically switch between point block Jacobi and multigrid preconditioners depending on the difficultly of the linear problem. We would also like to improve our solver so that it will scale linearly with the problem size by either using some sort of filtered smoothing, or other multigrid techniques used to compensate for lack of prolongation smoothing.

Chapter 5

Distributed Memory Bundle Adjustment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0016140. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

As with graph Laplacian problems, bundle adjustment problems are constantly growing in size. In [36], Google Street View uses billions of images and hundreds of billions of features to do planet-scale bundle adjustment and SfM. Improvements in core speeds and memory bandwidth can improve performance on single computers, but, to solve very large problems, using a distributed memory machine is necessary.

Surprisingly, the literature on bundle adjustment in distributed memory parallelism is limited. Eriksson et al. propose a novel formulation of the bundle adjustment optimization problem that is suitable to a distributed memory environment [26]. Zhang et al. extend this work with an improved partitioning scheme and the use of over-relaxation to improve convergence [76]. Kai et al. provide a parallel optimization framework, but do not provided any distributed memory results [51]. However, none of these papers take the approach of using distributed linear algebra to implement Levenberg-Marquardt.

We implemented Levenberg-Marquardt in distributed linear algebra using PETSc [10, 11]. Most of LM is easily adapted to distributed memory, but a couple parts pose a challenge. The first is distribution of cameras and points between processes. We choose to partition cameras evenly between processes, while points are replicated amongst all processes for which they are needed (those processes on which they is a

camera that views the point). We choose to distribute cameras over points so that the Hessian will be well balanced amongst processes. Once cameras and points are distributed, we can construct the local portions of the parameter vector, Jacobian, and approximate Hessian. The Jacobian is separated into a camera block and a point block, where each process contains entries for each camera or point it "owns."

Another difficulty arises in the construction of the Schur complement. Ceres Solver [4] sorts entries in the block of the Jacobian containing points and eliminates residuals interacting with the same point to quickly create the Schur complement. We cannot apply this same trick as residual blocks for a point may not reside on the same process as the camera that sees it, so our Schur complement construction is much slower. About 30 percent of solve time is spent in the Schur complement, but we leave improvements for future work.

We tested our implementation on NERSC's Cori: a Cray supercomputer with 36 "Haswell" Intel Xeon E5-2698 v3 cores per node and a Cray Aries interconnect. For each test, we run four MPI processes per physical node in order to obtain close to peak bandwidth. We used both point block Jacobi and GAMG preconditioners. GAMG—an algebraic multigrid implementation—is configured with unsmoothed prolongation, the near-nullspace of the bundle adjustment problem, and a Chebyshev point block Jacobi smoother. It does not use any of the specialized aggregation routines developed in the previous chapter. All of our results are for synthetic problems (see chapter 3) because there are no large real-world datasets available. Our datasets sizes range from 26 thousand cameras to 100 thousand cameras.

We tested both the weak and strong scaling of our implementation. Results are satisfactory, though there is no other distributed memory bundle adjuster to compare to. For strong scaling (figure 5.1), we see that our implementation maintains decent efficiency depending a little on the problem structure. The problem in the bottom right of figure 5.1 is less efficient because the linear problems are harder to solve. For comparison, this plot contains a serial solver data point (labeled Ceres Solver). Our solver requires 16 processes to surpass the serial solve. Most of the reason for the initial slowness of our solver is the lack of Schur complement trick. Still, our solver is at least two times faster than the serial solver with 32 processes (and it looks like would grow larger with more processes).

Our weak scaling results are a little more interesting. Using GAMG results in the scaling we would

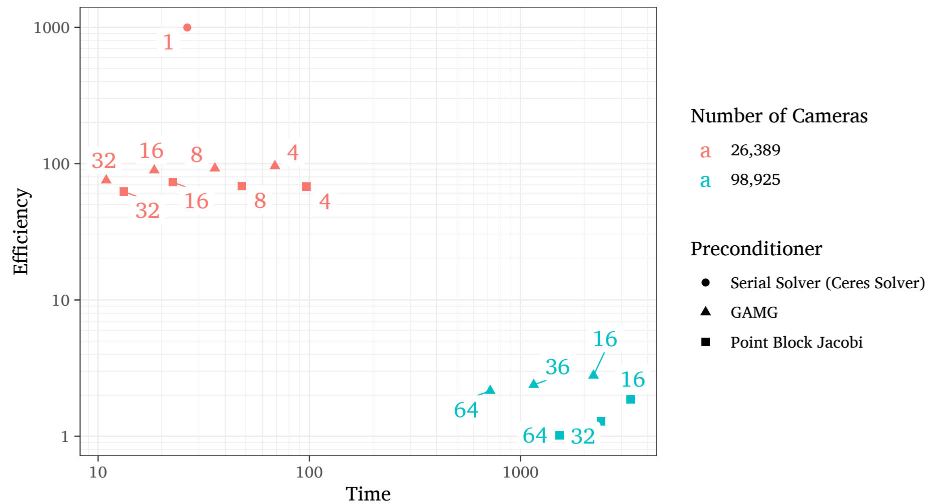


Figure 5.1: Loglog plot of strong scaling of Levenberg-Marquardt on a set of synthetic problems. The y-axis measures efficiency as $\frac{\text{\# of cameras}}{\text{time} \times \text{\# of nodes}}$. Horizontal lines on this plot indicate perfect scaling. Numeric labels indicate number of processes used.

expect: as problem sizes increase, efficiency slowly decreases. The point block Jacobi results seem to indicate some sort of bug in our solver. As the problem size increases, the total solve time decreases and the efficiency greatly increases. When we look at the iteration counts for these problems, we see them decreasing, which is contrary to what we see in the serial case (section 4.4.2). Overall, these weak and strong scaling results indicate that distributed linear algebra bundle adjustment is a good choice to solve large problem sets.

The initial results with our LM implementation indicate that distributed memory parallelism can be a quick way to solve large bundle adjustment problems. Scaling results are good, and the parallel solver can outperform serial solver when the problem can fit in memory. For large problems, our implementation allows for the solution of problems that are not feasible on a single node. Still, this is just an initial step. Work needs to be done to address the best data layout, how to handle more complicated camera models, how to handle parameters shared amongst many cameras, and how to distribute problems where to connectivity graph is dense.

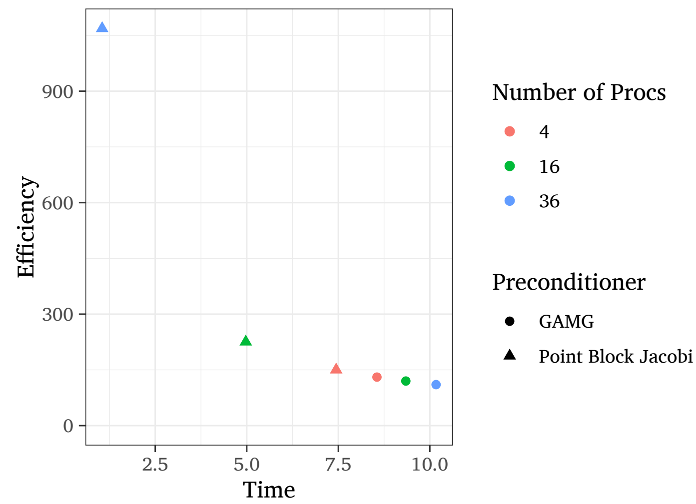


Figure 5.2: Loglog plot of weak scaling of Levenberg-Marquardt on a set of synthetic problems. The y-axis measures efficiency as $\frac{\text{\# of cameras}}{\text{time} \times \text{\# of nodes}}$. Horizontal lines on this plot indicate perfect scaling.

Chapter 6

Software Contributions

Performance results for new algorithms do no just spontaneously appear—they require an implementation. For all algorithms discussed in thesis, the implementation that generated results is provided. Making the code publicly accessible allows other researchers to verify results. Furthermore, these implementations offer something for new research to compare against and a way to find details not discussed in their respective papers. All code below is open source: it can be used and modified for any purpose.

- LigMG: Multigrid for graph Laplacians (chapter 2) is available at `https://github.com/ligmg/ligmg`.
- Bundle adjustment multigrid (chapters 4,5) is available at `https://gitlab.com/tkonolige/bundle_adjustment`.
- Synthetic graph generation (chapter 3) is available at `https://github.com/tkonolige/city2ba`.

I've also contributed to open source software projects when relevant:

- PyAMG: fixed algebraic strength-of-connection and implemented affinity strength-of-connection [46].
- CombBLAS: miscellaneous bug fixes and code modernization.
- PETSc: Levenberg-Marquardt implementation (chapter 5) and miscellaneous improvements.

Chapter 7

Conclusion

In this thesis, I have presented work on developing multigrid solvers for graph Laplacians in parallel environments and multigrid solvers for bundle adjustment in serial and parallel environments. The graph Laplacian solver features adaptations of low-degree elimination and Galerkin coarse grid construction to a distributed memory environment. Specifically, these operations are rephrased as linear algebraic operations so they can be used with a block-distributed matrix. My solver also features a novel parallel aggregation algorithm tuned for graphs with highly irregular connectivity. Scaling results show this distributed memory multigrid solver surpassing the state of the art serial solvers when using multiple processes (although serial results are slower than the state of the art due to trade offs for parallelism). This solver also shows good scalability dependent on problem size and number of processes.

I also developed a multigrid solver for bundle adjustment problems. The solver features a near-nullspace, strength-of-connection metric, aggregation algorithm, and smoother all targeted for bundle adjustment. The solver is up to 16 times faster than the state of the art on large, difficult problem sets. It shows superior scalability, so it is a good choice for problems that would be infeasible with conventional solvers. I have also provided some analysis of the difficulties existing solvers face on bundle adjustment problems. Ideally, my multigrid solver would be parallel, but work remains to be done to have it operate in distributed memory (specifically, the aggregation routine is inherently serial). To assess the validity of using distributed memory computing for bundle adjustment, I implemented a Levenberg-Marquardt nonlinear least-squares solver using PETSc. Using a partition of cameras then points, this implementation outperforms a serial

solver by 2x (although larger process numbers should be able to increase this speedup). Furthermore, this implementation can solve bundle adjustment problems that do not fit on a single node.

The results for bundle adjustment and graph Laplacian solvers are good, but, as always, there remains research to be done. For the graph Laplacian solver, it may be possible to adapt more sophisticated aggregation routines, such as DRA [49], to a parallel environment. On graphs that are not social networks, the current aggregation routine performs poorly. Having a metric to decide when this aggregation performs poorly would useful for determining when to use this solver. Another area that could result in significant speed up is applying graph sparsification between multigrid levels. Graph sparsification could also be a useful preprocessing step in bundle adjustment to reduce the complexity of forming the Schur complement, especially in situations where the camera-camera visibility graph becomes dense.

The bundle adjustment solver leaves some performance on the table. There are many implementation details that could be improved, but the most important improvement would be ensuring that the solver scales linearly. Applying techniques such as K-cycles [55] or block smoothing may bring the solver into a linear regime. Extending the solver to handle different bundle adjustment formulations also presents a direction for future research. It may be possible to compute the nullspace of a given formulation using automatic differentiation and user provided free modes.

Bibliography

[1] Mark Adams. Evaluation of three unstructured multigrid methods on 3d finite element problems in solid mechanics. International Journal for Numerical Methods in Engineering, 55(5):519–534, Oct 2002.

[2] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss-Seidel. Journal of Computational Physics, 188(2):593–610, 2003.

[3] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. Journal of Computational Physics, 188(2):593–610, Jul 2003.

[4] Sameer Agarwal, Keir Mierle, and Others. Ceres solver. http://ceres-solver.org.

[5] Sameer Agarwal, Noah Snavely, Steven M Seitz, and Richard Szeliski. Bundle adjustment in the large. In European Conference on Computer Vision, page 29–42. Springer, 2010.

[6] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications, 23(1):15–41, 2001.

[7] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. Quarterly of Applied Mathematics, 9(1):17–29, 1951.

[8] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In Proceedings of the 4th Annual ACM Web Science Conference, page 33–42. ACM, 2012.

[9] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. Multigrid smoothers for ultraparallel computing. SIAM Journal on Scientific Computing, 33(5):2864–2887, January 2011.

[10] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[11] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, Modern Software Tools in Scientific Computing, page 163–202. Birkhäuser Press, 1997.

[12] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing, 34(4):C123–C152, 2012.

[13] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. SIAM Review, 59(1):65–98, 2017.

[14] Paolo Boldi and Sebastiano Vigna. Four degrees of separation, really. In International Conference on Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM, page 1222–1227. IEEE, 2012.

[15] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013, page 1–12. IEEE, 2013.

[16] Erik G. Boman, Kevin Deweese, and John R. Gilbert. Evaluating the potential of a Laplacian linear solver. Computing Research Repository, abs/1505.00875, 2015.

[17] A. Brandt, J. Brannick, K. Kahl, and I. Livshits. Bootstrap AMG. SIAM Journal on Scientific Computing, 33(2):612–632, January 2011.

[18] Achi Brandt. General highly accurate algebraic coarsening. Electronic Transactions on Numerical Analysis, 10(1):21, 2000.

[19] Achi Brandt, James Brannick, Karsten Kahl, and Irene Livshits. Algebraic distance for anisotropic diffusion problems: multilevel results. Electronic Transactions on Numerical Analysis, 44:472–496, 2015.

[20] James Brannick, Yao Chen, Xiaozhe Hu, and Ludmil Zikatanov. Parallel unsmoothed aggregation algebraic multigrid algorithms on GPUs. In Numerical Solution of Partial Differential Equations: Theory, Algorithms, and Their Applications, page 81–102. Springer, 2013.

[21] Aydın Buluç and John R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. The International Journal of High Performance Computing Applications, 25(4):496 –509, 2011.

[22] Liang Chen. A modified Levenberg–Marquardt method with line search for nonlinear equations. Computational Optimization and Applications, 65(3):753–779, May 2016.

[23] Andrew J. Cleary, Robert D. Falgout, Van Emden Henson, and Jim E. Jones. Coarse-grid selection for parallel algebraic multigrid. In Solving Irregularly Structured Problems in Parallel, page 104–115. Springer Berlin Heidelberg, 1998.

[24] Tim Davis, Patrick Amestoy, David Bateman, Yanqing Chen, Iain Duff, Les Foster, William Hager, Scott Kolodziej, Stefan Larimore, Ekanathan Palamadai, Sivasankaran Rajamanickam, Sanjay Ranka, Wissam Sid-Lakhdar, and Nuri Yeralan. SuiteSparse, 2020.

[25] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software, 38(1):1:1–1:25, December 2011.

[26] Anders Eriksson, John Bastian, Tat-Jun Chin, and Mats Isaksson. A consensus-based framework for distributed bundle adjustment. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, page 1754–1762, 2016.

[27] Robert D. Falgout and Panayot S. Vassilevski. On generalizing the algebraic multigrid framework. SIAM Journal on Numerical Analysis, 42(4):1669–1693, Jan 2004.

[28] Emden R. Gansner, Yifan Hu, and Stephen North. A maxent-stress model for graph layout. IEEE Transactions on Visualization and Computer Graphics, 19(6):927–940, 2013.

[29] Michael W. Gee, Christopher M. Siefert, Jonathan J. Hu, Ray S. Tuminaro, and Marzio G. Sala. ML 5.0 smoothed aggregation user's guide. Technical report.

[30] Alan George and Joseph W. Liu. Computer Solution of Large Sparse Positive Definite Systems. Prentice-Hall, 1981.

[31] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. Applied Numerical Mathematics, 41(1):155–177, April 2002.

[32] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards, 49(6):409, December 1952.

[33] Yong-Dian Jian, Doru C. Balcan, and Frank Dellaert. Generalized subgraph preconditioners for large-scale bundle adjustment. In Outdoor and Large-Scale Real-World Scene Analysis, page 131–150, Berlin, Heidelberg, 2012. Springer.

[34] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen 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, STOC '13, page 911–920, New York, NY, USA, 2013. ACM.

[35] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José E. Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy G. Mattson. Mathematical foundations of the GraphBLAS. In High Performance Extreme Computing Conference (HPEC), 2016 IEEE, page 1–9. IEEE, 2016.

[36] Bryan Klingner, David Martin, and James Roseborough. Street view motion-from-structure-from-motion. In Proceedings of the IEEE International Conference on Computer Vision, page 953–960, 2013.

[37] Kurt Konolige. Sparse sparse bundle adjustment. In Proceedings of the British Machine Vision Conference, volume 10, page 102–1. Citeseer, 2010.

[38] Tristan Konolige and Jed Brown. A parallel solver for graph laplacians. In Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Ioannis Koutis, Gary L. Miller, and David Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. Computer Vision and Image Understanding, 115(12):1638 –1646, 2011. Special issue on Optimization for Vision, Graphics and Medical Imaging: Theory and Applications.

[40] Avanish Kushal and Sameer Agarwal. Visibility based preconditioning for bundle adjustment. In 2012 IEEE Conference on Computer Vision and Pattern Recognition, page 1442–1449. IEEE, 2012.

[41] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[42] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. Quarterly of Applied Mathematics, 2(2):164–168, 1944.

[43] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Transactions on Mathematical Software, 29(2):110–140, June 2003.

[44] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. SIAM Journal on Numerical Analysis, 16(2):346–358, Apr 1979.

[45] Oren E. Livne and Achi Brandt. Lean algebraic multigrid —fast graph Laplacian linear solver (source code), 2011.

[46] Oren E. Livne and Achi Brandt. Lean algebraic multigrid (LAMG): fast graph Laplacian linear solver. SIAM Journal on Scientific Computing, 34(4):B499–B522, 2012.

[47] Scott P. MacLachlan and Luke N. Olson. Theoretical bounds for algebraic multigrid performance: review and analysis. Numerical Linear Algebra with Applications, 21(2):194–220, February 2014.

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

[49] Artem Napov and Yvan Notay. An efficient multigrid method for graph Laplacian systems. Electronic Transactions on Numerical Analysis, 45:201–218, 2016.

[50] Stephen G. Nash and Ariela Sofer. Assessing a search direction within a truncated-newton method. Operations Research Letters, 9(4):219–221, Jul 1990.

[51] Kai Ni, Drew Steedly, and Frank Dellaert. Out-of-core bundle adjustment for large-scale 3d reconstruction. In 2007 IEEE 11th International Conference on Computer Vision, page 1–8. IEEE, 2007.

[52] Yvan Notay. Flexible conjugate gradients. SIAM Journal on Scientific Computing, 22(4):1444–1460, 2000.

[53] Yvan Notay. Aggregation-based algebraic multilevel preconditioning. SIAM Journal on Matrix Analysis and Applications, 27(4):998–1018, January 2006.

[54] Yvan Notay. An aggregation-based algebraic multigrid method. Electronic transactions on numerical analysis, 37(6):123–146, 2010.

[55] Yvan Notay and Panayot S. Vassilevski. Recursive Krylov-based multigrid cycles. Numerical Linear Algebra with Applications, 15(5):473–487, 2008.

[56] Luke N. Olson, Jacob Schroder, and Raymond S. Tuminaro. A new perspective on strength measures in algebraic multigrid. Numerical Linear Algebra with Applications, 17(4):713–733, 2010.

[57] Zhongnan Qu. Efficient Optimization for Robust Bundle Adjustment. PhD thesis, Technical University of Munich, Mar 2018.

[58] Dorit Ron, Ilya Safro, and Achi Brandt. Relaxation-based coarsening and multiscale graph organization. Multiscale Modeling & Simulation, 9(1):407–423, jan 2011.

[59] John W. Ruge and Klaus Stüben. Algebraic multigrid. In Steve McCormick, editor, Multigrid Methods, chapter 4, pages 73–130. SIAM, 1987.

[60] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing, 7(3):856–869, July 1986.

[61] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. Journal of Experimental Algorithmics (JEA), 19:1–24, 2015.

[62] Daniel A. Spielman. Spectral graph theory and its applications. In Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on, page 29–38. IEEE, 2007.

[63] Daniel A. Spielman. Algorithms, graph theory, and linear equations in Laplacian matrices. In Proceedings of the International Congress of Mathematicians, volume 4, page 2698–2722, 2010.

[64] Daniel A. Spielman et al. Laplacians.jl, 2017.

[65] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing, STOC '04, page 81–90, New York, NY, USA, 2004. ACM.

[66] Klaus Stüben. An introduction to algebraic multigrid. In Ulrich Trottenberg, Cornelis Oostrerlee, and Anton Schüller, editors, Multigrid, page 413–532. Elsevier Academic Press, San Diego, California, 2001.

[67] Rasmus Tamstorf, Toby Jones, and Stephen F. McCormick. Smoothed aggregation multigrid for cloth simulation. ACM Transactions on Graphics, 34(6):1–13, October 2015.

[68] Andrea Toselli and Olof B. Widlund. Domain decomposition methods—algorithms and theory. Springer Series in Computational Mathematics. Springer, 2005.

[69] Mark K. Transtrum and James P. Sethna. Improvements to the Levenberg-Marquardt algorithm for nonlinear least-squares minimization, 2012.

[70] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In International Workshop on Vision Algorithms, page 298–372. Springer, 1999.

[71] R.S. Tuminaro and C. Tong. Parallel smoothed aggregation multigrid : Aggregation strategies on massively parallel machines. In ACM/IEEE SC 2000 Conference (SC00). IEEE, 2000.

[72] Petr Vaněk, Jan Mandel, and Marian Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing, 56(3):179–196, 1996.

[73] Kyle Wilson and Noah Snavely. Robust global translations with 1DSFM. In Computer Vision — ECCV 2014, page 61–75. Springer, 2014.

[74] S. J. Wright and J. N. Holt. An inexact Levenberg-Marquardt method for large sparse nonlinear least squares. The Journal of the Australian Mathematical Society. Series B. Applied Mathematics, 26(4):387–403, April 1985.

[75] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. Knowledge and Information Systems, 42(1):181–213, 2015.

[76] Runze Zhang, Siyu Zhu, Tian Fang, and Long Quan. Distributed very large scale bundle adjustment by global camera consensus. In Proceedings of the IEEE International Conference on Computer Vision, page 29–38, 2017.