Sparse Sparse Bundle Adjustment

Kurt Konolige
http://www.willowgarage.com/~konolige

Willow Garage
68 Willow Road
Menlo Park, USA

Abstract

Sparse Bundle Adjustment (SBA) is a method for simultaneously optimizing a set of camera poses and visible points. It exploits the sparse primary structure of the problem, where connections exist just between points and cameras. In this paper, we implement an efficient version of SBA for systems where the secondary structure (relations among cameras) is also sparse. The method, which we call Sparse SBA (sSBA), integrates an efficient method for setting up the linear subproblem with recent advances in direct sparse Cholesky solvers. sSBA outperforms the current SBA standard implementation on datasets with sparse secondary structure by at least an order of magnitude, while also being more efficient on dense datasets.

1 Introduction

Sparse Bundle Adjustment (SBA) is the standard method for optimizing a structure-from-motion problem in computer vision. With the success of Photosynth and similar systems for stitching together large collections of images [16], attention has turned to the problem of making SBA more efficient. There are two different types of large-scale systems:

- Photosynth-type systems focus on reconstruction from a large number of images concentrated in a small area; we call these *object-centered*.

- Visual mapping systems [1, 3, 9] cover a more extended area with fewer images, and real-time performance is often important (see Figure 1).

These types are at two ends of a spectrum: object-centered systems produce dense relations between cameras, while visual mapping systems are much sparser, with cameras in a local neighborhood sharing common points (Guilbert et al. call these "sparse systems" [8]). In this paper, we are interested in fast SBA methods for the latter case, where it is possible to exploit the sparse secondary structure (camera to camera relations) of the problem.

Nonlinear optimization in SBA typically proceeds by iteration: form a linear subproblem around the current solution, solve it, and repeat until convergence. For large problems, the computational bottleneck is usually the solution of the linear subproblem, which can grow as the cube of the number of cameras. The fill-in of the linear problem is directly tied to the camera-point structure of the problem: if each camera only sees features in a small neighborhood of other cameras, the number of non-zero elements grows only linearly or nearly linearly with the number of cameras.
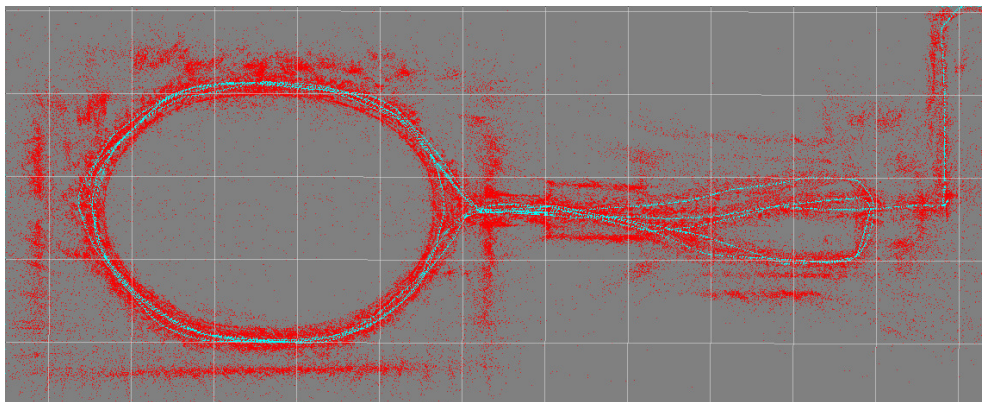
Figure 1: An overhead view showing of the New College mapping dataset [15], with 2.2k views (cyan) and 290k points (red). Grid lines are at 10 m. At the end, sSBA computes each iteration in 6 seconds.

In this paper we present an engineering approach to efficiently solving SBA problems with sparse linear subproblems. Our approach is to exploit recent fast direct Cholesky decomposition methods to solve the linear subproblem. These methods use a compressed representation of large sparse matrices, and we present a method for efficiently handling the block data structures of SBA to take advantage of this representation. The end result is a system, which we call *Sparse SBA* or sSBA, that outperforms the current reference system for SBA from Lourakis and Argyros [11] by an order of magnitude on sparse seconary-structure problems, and uses far less space on large problems. For example, Figure 1 shows a reconstruction of a New College dataset [15] that contains 2200 views and 290k points, and is solved in about 6 seconds per iteration at the end. Interestingly, sSBA also outperforms [11] on problems with dense secondary structure, where the setup computation often dominates, although by a lesser margin.

2   Related Work

The standard reference for SBA is the monograph of Triggs et al. [17]. This work explores many of the mathematical and computational aspects of SBA, including various methods for solving the nonlinear optimization problem at the heart of SBA. In this paper, we use Levenberg-Marquardt [14], which is a standard algorithm for solving unconstrained nonlinear optimization. Alternative solvers for SBA include preconditioned conjugate gradient [2] or Powell's dog-leg solver [12].

The LM implementation of SBA repeatedly solves a large linear subproblem whose LHS matrix is positive definite. Recent work in direct sparse Cholesky solvers [4] has yielded algorithms that are very efficient for large problems, and we use these methods here.

Most current applications that incorporate SBA use an open-source version developed by Lourakis and Argyros [11], which we call *laSBA*. For example, the open source Bundler program originally used for PhotoSynth [16], an application for stitching together tourist photos, uses laSBA as its optimization engine. An exception is the work of Klein and Murray [9], which has an SBA engine. We have tested this system and found it slower than laSBA,

so laSBA will be our reference implementation. In the context of mapping systems, there are references to using sparse solvers in SBA, e.g., Guilbert et al. [8] mention sparse QR decomposition and supernodal Cholesky methods. However, there is no explicit algorithm for setting up the sparse linear problem, which we have found to be an important bottleneck.

3 SBA Basics

This section summarizes the basic formulation of Sparse Bundle Adjustment used in the paper. For the most part it follows the excellent exposition of Engels et al. [5], and the reader can consult this paper for derivations.

3.1 Error Formulation

Sparse Bundle Adjustment (SBA) is a method of nonlinear optimization among camera frames (c_i) and points (p_j). Each camera frame consists of a translation t_i and rotation R_i giving the position and orientation of the frame in global coordinates. For any such pair, the *measured* projection of p_j on the camera frame is called $\bar{z}_{ij}$. The *calculated* feature value comes from the projection equation:

$$g(c_i, p_j) \equiv R_i^\top (t_i - p_j)$$
$$h(c_i, p_j) \equiv g_{x,y}(c_i, p_j)/g_z(c_i, p_j) \tag{1}$$

The function g transforms the point p_j into c_i's coordinate system, and h projects it to a normalized image plane.

The error function associated with a projection is the difference between the calculated and measured projection. The total error is the sum over all projections.

$$e_{ij} \equiv h(c_i, p_j) - \bar{z}_{ij}$$
$$E(\mathbf{c}, \mathbf{p}) \equiv \sum_{ij} e_{ij}^\top \Lambda_{ij} e_{ij} \tag{2}$$

Λ_{ij} is the precision matrix (inverse covariance) of the feature measurement. In the case of SBA, it is often assumed to be isotropic (diagonal) and on the order of a pixel, making it the identity matrix. For simplicity we drop it from the rest of the exposition; the system could be easily modified to accommodate it.

3.2 Levenberg-Marquardt System

The optimal placement of $\mathbf{c}, \mathbf{p}$ is found by minimizing the total error in Equation 2. A standard method for solving this problem is to iterate a linearized solution around the current values of $\mathbf{c}, \mathbf{p}$. The linear solution is found by second-order Taylor expansion around $\mathbf{c}, \mathbf{p}$, and an approximation of the second-order derivative matrix (the Hessian) by Jacobian products (the *normal* or Gauss-Newton approximation).

The resultant linear system is formed by stacking the variables $\mathbf{c}, \mathbf{p}$ into a vector $\mathbf{x}$, and the error functions into a vector $\mathbf{e}$. Let

$$\mathbf{J} \equiv \frac{\partial \mathbf{e}}{\partial \mathbf{x}}$$

$$\mathbf{H} \equiv \mathbf{J}^\top \mathbf{J} \tag{3}$$

The linear system is:

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{J}^\top\mathbf{e} \tag{4}$$

In general, solving this system is not guaranteed to produce a step $\Delta\mathbf{x}$ that decreases the error. The Levenberg-Marquardt (LM) method augments $\mathbf{H}$ by adding $\lambda\,\mathrm{diag}(\mathbf{H})$, where λ is a small positive multiplier. Larger λ produces a gradient descent system that will converge (but slowly). There are various strategies for manipulating λ, initially using gradient descent and then the faster Newton-Euler method at the end.

3.3 Primary Structure

The size of the matrix $\mathbf{H}$ is dominated by $||\mathbf{p}||$, which in typical problems is several orders of magnitude larger than $||\mathbf{c}||$. But we can take advantage of the structure of the Jacobians to reduce Equation 4 to just the variables $\mathbf{c}$. If we organize Equation 4 so that cameras and points are clustered, it has a characteristic structure:

$$\begin{bmatrix} \mathbf{J}_\mathbf{c}^\top\mathbf{J}_\mathbf{c} & \mathbf{J}_\mathbf{c}^\top\mathbf{J}_\mathbf{p} \\ \mathbf{J}_\mathbf{p}^\top\mathbf{J}_\mathbf{c} & \mathbf{J}_\mathbf{p}^\top\mathbf{J}_\mathbf{p} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{c} \\ \Delta\mathbf{p} \end{bmatrix} = \begin{bmatrix} -\mathbf{J}_\mathbf{c}^\top\mathbf{e}_\mathbf{c} \\ -\mathbf{J}_\mathbf{p}^\top\mathbf{e}_\mathbf{p} \end{bmatrix} \tag{5}$$

where $\mathbf{J}_\mathbf{c}$ is the Jacobian with respect to camera variables, and $\mathbf{J}_\mathbf{p}$ with respect to point variables. Because all the error functions involve one camera and one point, the Jacobian products $\mathbf{J}_\mathbf{c}^\top\mathbf{J}_\mathbf{c}$ and $\mathbf{J}_\mathbf{p}^\top\mathbf{J}_\mathbf{p}$ are block-diagonal. After some manipulation, the reduced system is

$$[\mathbf{H}_{\mathbf{cc}} - \mathbf{H}_{\mathbf{cp}}\mathbf{H}_{\mathbf{pp}}^{-1}\mathbf{H}_{\mathbf{pc}}]\Delta\mathbf{c} = -(\mathbf{J}_\mathbf{c}^\top\mathbf{e}_\mathbf{c} - \mathbf{J}_\mathbf{p}^\top\mathbf{H}_{\mathbf{cp}}\mathbf{H}_{\mathbf{pp}}^{-1}\mathbf{e}_\mathbf{p}) \tag{6}$$

where $\mathbf{H}_{xy}$ refers to the Jacobian products in Equation 5. Note the matrix inversion is simple because of the block-diagonal structure of $\mathbf{H}_{\mathbf{pp}}$.

Solving this equation produces an increment $\Delta\mathbf{c}$ that adjusts the camera variables, and then is used to update the point variables according to

$$\Delta\mathbf{p} = -\mathbf{H}_{\mathbf{pp}}^{-1}(\mathbf{J}_\mathbf{p}^\top\mathbf{e}_\mathbf{p} + \mathbf{H}_{\mathbf{pc}}\Delta\mathbf{c}) \tag{7}$$

In forming the left-hand side of Equation 6, the main computational bottleneck is computing the product $\mathbf{H}_{\mathbf{cp}}\mathbf{H}_{\mathbf{pp}}^{-1}\mathbf{H}_{\mathbf{pc}}$. For any given point p, if p projects onto n cameras (its *track length*, then this product has $n(n-1)$ additions to the left-hand matrix. If the average point track length grows linearly with the size of the system, then the effort to set up the system grow quadratically. On the other hand, if the average track size is constant, it grows only linearly.

4 Sparse Linear Systems

We are interested in large systems, where the number of camera variables $||\mathbf{c}||$ can be 10k or more (the largest real-world dataset we have used is about 3k poses, but we can generate synthetic datasets of any order). The number of system variables is $6\cdot||\mathbf{c}||$, and the reduced system matrix of Equation 6 has size $36\cdot||\mathbf{c}||^2$, or over 10^9 elements. Manipulating such large matrices is expensive. To do it efficiently, we have to take advantage of its sparse structure. The sparsity pattern of the reduced SBA system is referred to as *secondary structure*.

The solution of Equation 6 has two computational intensive parts.
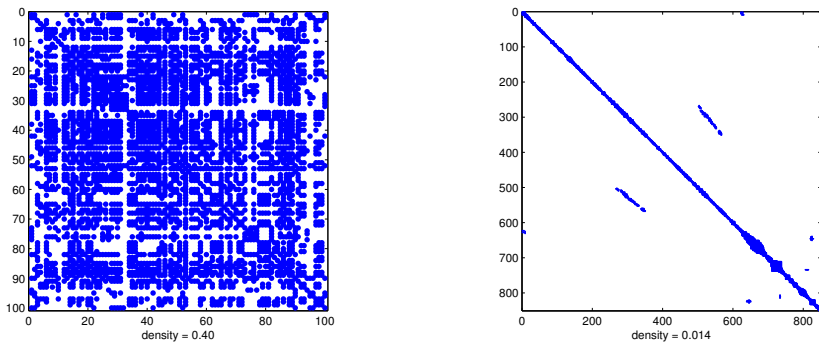
1. **H** matrix construction.

Figure 2: **H** matrix non-zero patterns for the Venice dataset (left) and the Intel indoor dataset (right). Only the first 100 frames (out of 871) of the Venice dataset are diagrammed, because it would be too dense to show the full structure.

2. **H** matrix solution.

Note that back-substitution and the construction of the RHS of Equation 6 are only a minor contribution, and we ignore them for now. For solving the matrix equation, we rely on available direct Cholesky decomposition for sparse matrices, described below. Matrix construction involves forming the product $\mathbf{H_{cp}H_{pp}^{-1}H_{pc}}$, and in subsequent sections we show how to do this efficiently, so that the structures necessary for sparse Cholesky decomposition are easily generated.

4.1   Sparse Secondary Structure

As discussed in the introduction, there are two typical usage patterns for SBA. In one, a set of images are taken of an object, so most of the images have features in common and the secondary structure is dense. For example, in the Venice dataset of tourist photos[1], the density of **H** is 40%, that is, non-zeros account for 40% of the matrix entries.

On the other end of the spectrum are datasets from visual mapping, where usually a single camera moves around an area, and the images are registered to produce an extended map. For example, in the Intel Seattle indoor dataset[2], the camera motion is mostly along corridors, and the density is only 1.4%. The difference in the sparsity pattern is shown in Figure 2. Note that the mapping pattern consists of a fat diagonal band, with some parallel bands for overlapping trajectories.

There are also scenarios that are in-between these two, for example, the datasets from the Samantha project [6] show density between 10% and 17%, as they cover extended outside areas.

---

[1]This dataset provided courtesy of Noah Snavely.
[2]This dataset provided courtesy of Peter Henry.

4.2　Compressed Column Storage

Many sparse matrix methods use *compressed column storage* (CCS) format for representing matrices. The figure below shows the basic idea.

$$
\begin{bmatrix}
1 & 0 & 4 & 0 \\
0 & 5 & 0 & 2 \\
0 & 0 & 0 & 1 \\
6 & 8 & 0 & 0
\end{bmatrix}
\Rightarrow
$$

col_ptr	0		2		4	5		7
row_ind	0	3	1	3	0	1	2	
val	1	6	5	8	4	2	1	

(8)

Each nonzero entry in the array is placed in the *val* vector. Entries are ordered by column first, and then by row within the column. *col_ptr* has one entry for each column, plus a last entry which is the number of total nonzeros (*nnz*). The *col_ptr* entry for a column points to the start of the column in the *row_ind* and *val* variables. Finally, *row_ind* gives the row index of each entry within a column.

CCS format is storage-efficient, but is difficult to create incrementally, since each new nonzero addition to a column causes a shift in all subsequent entries. Trying to create it directly from the product $\mathbf{H_{cp}H_{pp}^{-1}H_{pc}}$ would be inefficient. Instead, we first create a parallel column-oriented block structure, and then transfer this structure to the sparse matrix format.

4.3　Block-oriented Sparse Matrix Creation

The most compute-intensive part of creating $\mathbf{H}$ involves an outer product over the projections in each point track (for details of the whole algorithm, see Engels et al. [5]). Assume the cameras on the track are ordered. Consider the track of given point p. For each camera c on the track, do the following:

1. Form the product $T_{pc} = J_c^\top J_p (J_p^\top J_p)^{-1}$.

2. For each camera $c' \geq c$ on the track, subtract $T_{pc} J_p^\top J_{c'}$ from the 6x6 block (c, c') of $\mathbf{H}$.

In our version of this algorithm, we create a sparse structure for accessing arbitrary 6x6 blocks i, j of $\mathbf{H}$; since the matrix decomposition only uses the upper triangular part of $\mathbf{H}$, we have $i \leq j$. There are two requirements for this structure: it should have fast random access, and it should be navigable by column in row order for the creation of the CCS format matrix. These are conflicting requirements – for example, a hash table would give fast random access, but does not allow navigation by column.

Our approach is to use a C++ `std::map` container for each column of $\mathbf{H}$. The `map` is keyed by row index, and its value is the 6x6 block. Lookup of an arbitrary row element within a `map` is order $\log n$ in the number of elements in the map, while column lookup is constant time (simple array access).

An important property of `map` is that it is ordered by its key, for efficient access to blocks ordered by row. Once the block data structure is created by running through all the tracks, we use the ordered nature of the maps to create the sparse CCS format of $\mathbf{H}$ by looping over each map in the order of its keys, first to create the column and row indices, and then to put in the values. The reason for separating the column/row creation from value insertion is because the former only has to be done once for any set of iterations of LM.

4.4 Complexity

The computational complexity for forming the $\mathbf{H}$ matrix depends on the average track size. In object-centered systems, the track size grows linearly with the number of frames N, so each track is quadratic in N. The number of tracks stays constant or increases only slowly, since each new frame is connected to existing tracks; hence the complexity is order N^2. Finally, the cost of insertion grows as $\log N$, hence the total cost is $N^2 \log N$.

If the constraints are sparse as in mapping, the track size is bounded, so the computation for each track is constant. The number of tracks grows linearly with N, since new tracks appear at a constant rate. Insertion cost is constant, because the average `map` size is constant. The total complexity is thus order N, which is lower than for object-centered systems.

4.5 Sparse Linear Systems

For solving (4) in sparse format, we use the CHOLMOD package [2]. This package has a highly-optimized Cholesky decomposition solver for sparse linear systems. It employs several strategies to decompose $\mathbf{H}$ efficiently, including a logical ordering and an approximate minimal degree (AMD) algorithm to reorder variables when $\mathbf{H}$ is large.

In general the complexity of decomposition will be $O(n^3)$ in the number of variables. For sparse matrices, the complexity will depend on the density of the Cholesky factor, which in turn depends on the structure of $\mathbf{H}$ and the order of its variables. Mahon et al. [13] have analyzed the behavior of the Cholesky factorization as a function of the loop closures in the SLAM system. If the number of loop closures is constant, then the Cholesky factor density is $O(n)$, and decomposition is $O(n)$. If the number of loop closures grows linearly with the number of variables, then the Cholesky factor density grows as $O(n^2)$ and decomposition is $O(n^3)$.

5 Experiments

To exercise sSBA, we performed experiments on both synthetic and real datasets. With synthetic datasets, it is possible to perform extensive experiments and to isolate the effect of variables on performance. Real datasets verify the conclusions of the synthetic datasets, and show the system functioning in real-world situations.

5.1 Lourakis and Argyros SBA

In the experiments, we compared sSBA against the system of Lourakis and Argyros [11] (laSBA), which is the standard open-source SBA system in the vision community. laSBA performs the same operations as sSBA: $\mathbf{H}$-matrix formation, $\mathbf{H}$-matrix solution, and back-substitution. Like sSBA, it uses unit quaternions and local angle representations. Finally, similar to sSBA, laSBA stores only non-zero H_{ij} blocks. The major differences between laSBA and sSBA are:

1. laSBA uses a compressed row storage (CRS) format for indexing, rather than the `map` data structure.

2. laSBA does not use the track-oriented algorithm for decomposing $\mathbf{H_{cp}H_{pp}^{-1}H_{pc}}$ in forming $\mathbf{H}$.
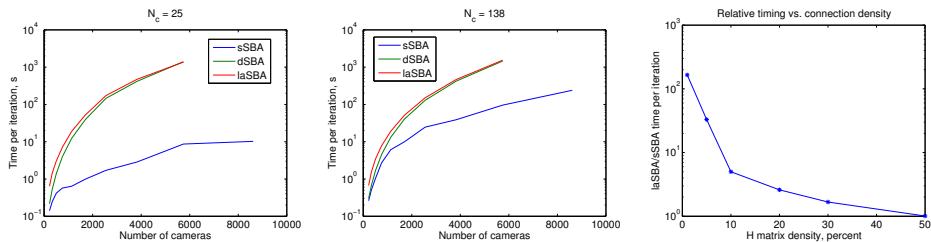
Figure 3: First two plots: time per iteration for different connection densities, over laSBA, dSBA, and sSBA algorithms. N_c is the average number of connections each camera has to other cameras. Third plot: relative time of laSBA vs. sSBA for varying **H** density. Note Y-axis log scale on all graphs.

3. laSBA constructs a dense matrix **H** from the H_{ij} blocks, and uses LAPACK Cholesky decomposition for its solution.

To make the comparison more fair, we substituted a faster dense Cholesky decomposition, using Cholesky LLT (non-pivoting) from the Eigen package development branch.[3] This routine is several times faster than LAPACK (with Atlas BLAS) on the tested machines. We used the fastest version of laSBA, with expert drivers and analytic Jacobians.

In addition to sSBA, we tested a variant, called dSBA, that constructs and solves a dense **H** matrix. This variant uses direct access to the **H** matrix, rather than constructing an intermediate structure for the H_{ij} blocks.

All experiments were run on the same machine using a single core, an Intel i7 with 8 MB of primary cache and 49 GB of main memory, running at 3.0 GHz. We report time per iteration of the LM method in the following sections, rather than time to convergence, which depends on the parameters of the LM iteration.

5.2   Spiral Trajectory

In the synthetic datasets, the camera points and moves forward along a spiral trajectory. By varying the density of points and the range of the camera, it is possible to create datasets with different secondary structure. In the first set of experiments, we kept the number of projections per camera at around 500, while varying connection strength. We used two measures of connection.

1. Average number of connections per camera (N_c).

2. Density of the **H** matrix.

The first measure represents typical mapping scenarios, where a camera is connected to its neighbors in an extended environment (including loop closure neighbors). The second measure is more appropriate for object-centered datasets, where adding views raises the number of connections per camera.

Examing the first two plots for mapping in Figure 3, the trend of sSBA vs. laSBA timings is clear. For sparse connections ($N_c = 25$ per camera, average), sSBA is about 7 times faster for small number of views, and over 100 times faster at 6000 views. Beyond this, laSBA
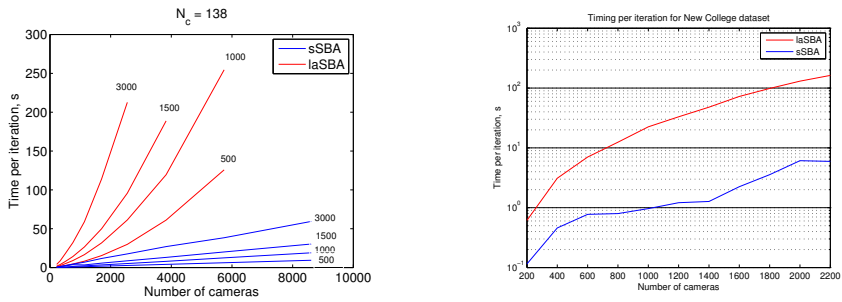
---

[3]http://eigen.tuxfamily.org

Figure 4: Left: setup time of sSBA vs. laSBA, with fixed camera connections but varying projections. The curves show setup time for 500, 1000, 1500, and 3000 point projections per camera. Right: relative timing for laSBA and sSBA on the New College dataset.

runs out of memory. Note that sSBA trends linearly for large number of views, as expected from the constant number of camera connections in the **H** matrix. An interesting effect is that dSBA out-performs laSBA for smaller datasets. Since they both use the same Cholesky decomposition, the difference arises from the setup of the **H** matrix. Note that this difference can be quite significant, with dSBA being 4 times as fast for 225 views.

For denser connections, that advantage of sSBA diminishes, although it is still substantial. In the middle plot, $N_c = 138$, sSBA is faster than laSBA by about 4 times for the smallest dataset, up to 60 times at 6000 views. In both these datasets, dense **H** systems ran out of memory for the largest graphs.

The third figure shows the relative timing of sSBA vs. laSBA for different densities of **H**. For low densities, there is an obvious advantage to using sparse decomposition. For higher densities, it is surprising that sSBA performs as well as laSBA, since the overhead in manipulating the sparse matrix format grows as it fills up – with 64-bit integers, a sparse matrix in CCS format uses as more space then a dense matrix. One reason sSBA performs so well is that the matrix construction step is more efficient that laSBA (recall they use different block-oriented sparse representations in the construction phase). For a constant $N_c = 138$ (moderate density), track lengths are constant over problem size, and setup times should be linear. But in Figure 4, laSBA has a quadratic trend with problem size, while sSBA is nearly linear. This trend holds for different average projection count per camera, and is an inefficiency in the laSBA algorithm independent of the density of **H**.

5.3 Real-World Datasets

We examined three different real-world datasets: Venice, a Photosynth collection[4]; Samantha, a set of three single-camera collections from the University of Verona[5], and an indoor dataset from Intel Seattle mentioned earlier. Table 1 has timings for sSBA and laSBA. Venice is an object-centered dataset with 40% **H** fill. Because of its sparse block construction, sSBA is slower in solving **H**, but much faster at constructing it; overal sSBA is 4 times faster. The Samantha datasets are intermediate between object-centered and mapping types; here sSBA does much better in solving **H**. Finally, the Intel dataset is an indoor mapping sequence with only 1.4% density, and sSBA is much faster than laSBA, by a factor of 80.

---

[4]Courtesy of Noah Snavely.
[5]http://profs.sci.univr.it/ fusiello/demo/samantha/

Name	# cams	projs/cam	**H** fill	setup	solve	total
Venice	871	3259	40%			
sSBA				12.15	6.09	**18.24**
laSBA				79.58	5.30	84.88
Samantha Erbe	184	705	17%			
sSBA				0.13	0.017	**0.15**
laSBA				1.0	0.8	1.1
Samantha Bra	320	1222	10%			
sSBA				0.40	0.028	**0.43**
laSBA				4.47	0.38	4.8
Intel	851	133	1.4%			
sSBA				0.079	0.023	**0.10**
laSBA				1.91	6.10	8.01

Table 1: Time per iteration (in seconds) for several real-world datasets.

sSBA is used as the back end of a full VSLAM system available in ROS [10]. Figure 1 shows the result of performing several iterations every 10 keyframes on the New College dataset [15]. sSBA runs in a separate process; it is efficient enough to perform full bundle adjustment through about 2k frames in real time, with complex cross-connections (Figure 4.

5.4   Limitations of sSBA

Currently we have implementations of sSBA for both stereo and monocular pinhole cameras that are calibrated. We intend to add a robust cost measure, and the ability to estimate camera focal length and distortion. laSBA already incorporates the latter feature, and is a more general system, allowing user-define "drivers" for different camera types. Another proposed addition is the use of "smoothing priors" [7], which help with stability, especially in monocular systems.

6   Conclusions

sSBA is a system that takes advantage of sparse secondary structure in the SBA problem to perform efficient optimization. It constructs a sparse Hessian matrix using sparse ordered storage for its sub-blocks. sSBA outperforms the standard SBA implementation (laSBA) by over an order of magnitude on typical mapping datasets, which have sparse secondary structure. It is also faster than laSBA on more dense datasets, up to 50% fill-in on the Hessian. The code will be available open-source with a BSD license.

References

[1] Motilal Agrawal and Kurt Konolige. FrameSLAM: From bundle adjustment to real-time visual mapping. *IEEE Transactions on Robotics*, 24(5), October 2008.

[2] M. Byrod and K. Astrom. Bundle adjustment using conjugate gradients with multiscale programming. In *British Machine Vision Conference*, 2009.

[3] Mark Cummins and Paul M. Newman. Highly scalable appearance-only SLAM – FAB-MAP 2.0. In *Robotics Science and Systems*, 2009.

[4] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[5] Chris Engels, Henrik Stewénius, and David Nister. Bundle adjustment rules. *Photogrammetric Computer Vision*, September 2006.

[6] M. Farenzena, A. Fusiello, and R. Gherardi. Structure-and-motion pipeline on a hierarchical cluster tree. In *ICCV Workshop on 3-D Digital Imaging and Modeling*, pages 1489–1496, 2009.

[7] Michela Farenzena, Adrien Bartoli, and Youcef Mezouar. Efficient camera smoothing in sequential structure-from-motion using approximate cross-validation. In *ECCV (3)*, pages 196–209, 2008.

[8] Nicolas Guilbert, Adrien Bartoli, and Anders Heyden. Affine approximation for direct batch recovery of euclidian structure and motion from sparse data. *Int. J. Comput. Vision*, 69(3):317–333, 2006.

[9] Georg Klein and David Murray. Parallel tracking and mapping for small AR workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan, November 2007.

[10] K. Konolige. Vslam package. Willow Garage, Robot Operating System, WG-ROS-PKG, SVN Repository, 2010. Available online at https://code.ros.org/svn/wg-ros-pkg/trunk/vision/vslam_system.

[11] M.I. A. Lourakis and A.A. Argyros. SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Trans. Math. Software*, 36(1):1–30, 2009.

[12] M.I.A. Lourakis and A.A. Argyros. Is levenberg-marquardt the most efficient optimization algorithm for implementing bundle adjustment? In *Proc. International Conference on Computer Vision*, 2005.

[13] I.J. Mahon, S.B. Williams, O. Pizarro, and M. Johnson-Roberson. Efficient view-based SLAM using visual loop closures. *IEEE Transactions on Robotics*, 24(5):1002–1014, October 2008.

[14] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics*, 11(2):431–441, 1963.

[15] M. Smith, I. Baldwin, W. Churchill, R. Paul, and P. Newman. The new college vision and laser data set. *International Journal for Robotics Research (IJRR)*, 28(5):595–599, May 2009. ISSN 0278-3649.

[16] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Skeletal sets for efficient structure from motion. In *Conference on Computer Vision and Pattern Recognition*, 2008.

[17] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzibbon. Bundle adjustment - a modern synthesis. In *Vision Algorithms: Theory and Practice*, LNCS, pages 298–375. Springer Verlag, 2000.