Efficient Implementation for Block Matrix Operations for Nonlinear Least Squares Problems in Robotic Applications

Lukas Polok, Marek Solony, Viorela Ila, Pavel Smrz and Pavel Zemcik

Abstract— **A large number of robotic, computer vision and computer graphics applications rely on efficiently solving the associated sparse linear systems. Simultaneous localization and mapping (SLAM), structure from motion (SfM), non-rigid shape recovery, and elastodynamic simulations are only few examples in this direction. In general, these problems are nonlinear and the solution can be approximated by incrementally solving a series of linearized problems. In some applications, the size of the system considerably affects the performance, especially when the sparsity is low. This paper exploits the block structure of such problems and offers very efficient solutions to manipulate block matrices within iterative nonlinear solvers. The resulting method considerably speeds-up the execution of the implementation of the nonlinear optimization problem. In this work, in particular, we focus our effort on testing the method on SLAM applications, but the applicability of the technique remains general. Our implementation outperforms the state of the art SLAM implementations on all tested datasets. In incremental mode, where a larger portion of time is spent in updating the system, our implementation is on average two times faster than the others.**

I. INTRODUCTION

In robotics, simultaneous localization and mapping (SLAM) is often formulated as a nonlinear least squares problem. Similar problems such as structure from motion (SfM) in computer vision [1] or elastodynamic simulations in computer graphics [2] rely on solving large nonlinear systems. Efficient incremental online algorithms for solving the underlying nonlinear least square problem are essential in online, real applications. The types of problems we are targeting in this paper consist of repeatedly assembling and solving large, positive-definite, sparse linear systems. The initial problem is nonlinear, and it is usualy addressed by iteratively solving a sequence of linear systems. The most computationally demanding part is to assemble and solve the linearized system at each iteration.

The linear system can be solved either using direct or iterative methods. Direct methods, such as Cholesky or QR factorizations, are based on repeatedly factorizing a large matrix and backsubstitution to obtain the solution. Iterative methods, such as conjugate gradient, on the other hand, iteratively approximate the solution of the linear system. Iterative methods are more efficient from the storage (memory) point of view, since they only require access to the gradient, but they can suffer from poor convergence. Direct methods produce more accurate solutions and avoid convergence difficulties but they typically require a lot of storage as well as efficient elimination orderings to be found in order to maintain the sparsity of the resulting factors.

The authors are with Brno University of Technology, Faculty of Information Technology. Bozetechova 2, 612 66 Brno, Czech Republic. {ipolok,isolony,ila,smrz,zemcik}@fit.vutbr.cz

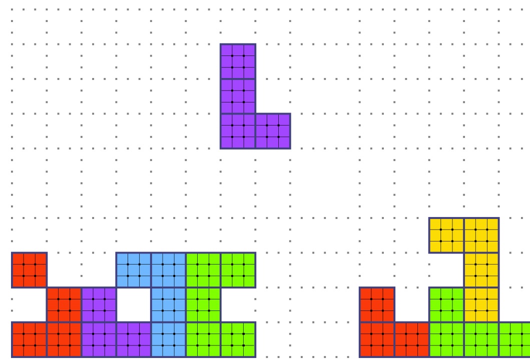


Fig. 1. An example of a randomly generated sparse block matrix composed of 31 blocks, 3^2 elements each, used in testing operations on block matrices.

In robotics, approaching SLAM as a nonlinear optimization on graphs showed to provide very efficient solutions to moderate scale and well-behaved SLAM applications [3], [4], [5], [6], [7]. Graphs allow more natural representation of nonlinear least squares problems such as SLAM, where we estimate a set of variables such as the robot poses and/or landmarks position, given a set of measurement constraints between those variables. The goal is to find the optimal configuration of the variables that maximally satisfy the set of nonlinear constraints. The existing methods repeatedly solve a sequence of linear systems in an iterative Gauss-Newton (GN) or Levenberg-Marquardt (LN) nonlinear solver. Real applications such as online mapping and localization of a robot in a large area and over very long periods of time require extremely fast methods for building, updating and solving the sequence of linearized systems. It involves operating on matrices having a block structure, where the size of the blocks corresponds to the number of degrees of freedom of the variables.

Some of the existing implementations rely on sparse block-structure schemes [8], [7]. The block structure is maintained until the point of solving the linear system. Here is where CSparse [9] or CHOLMOD [10] libraries are used to perform the matrix factorization. Those are state of the art element-wise implementation of operations on sparse matrices.

The advantage of *element-wise* sparse matrix schemes is that the arithmetic operations can be performed efficiently. Compressed column storage (CCS) format used in CSparse is an efficient way to store the sparse data in the memory. The

disadvantage of this format is its inability or impracticality to change matrix structurally or numerically once it has been compressed. The *block-wise* schemes are complementary, their advantages include both easy numeric and structural matrix modification, at the cost of memory overhead and reduced arithmetic efficiency, speed-wise.

In the SLAM context, the linear system is updated every iteration of the GN or LM solver. This involves repeatedly building the corresponding matrix. The well known element-wise CCS sparse matrix representation is as efficient (sometimes even more efficient) as any block matrix structure, in the case of operating on a set of structurally-different matrices. The SLAM problem, however, involves operating iteratively on matrices where large parts of the matrix have the same structure. In such case, block matrix schemes can be very proficient, as they allow for modifying the block structure as well as efficiently modifying the numeric content.

We propose a fast and cache efficient data structure for sparse block matrix representation, which combines the advantages of element-wise and block-wise schemes. It enables simple matrix modification, be it structural or numerical, while also maintaining, and sometimes even exceeding the speed of element-wise operations schemes. In this direction, we introduce a new storage model for block matrices, which improves the memory coherency and the cache utilization. Fig 1 shows an example of a random generated block, sparse matrix, we used in testing the arithmetic operations using the proposed storage scheme. Another important advantage is the overall robustness of the structure, allowing for error-checking, resulting in easier usage and in more stable error-free code.

This paper provides a new scheme for efficient sparse block matrix operations, which are the core operations for solving many least squares problems. The new block matrix scheme has been integrated in standard nonlinear least squares algorithms, and tested on several well-known SLAM datasets. The experimental evaluation shows that, even without any algorithmic improvements, the proposed methodology provides important speedups, in both batch and incremental settings.

The paper is structured as follows. In the next section we comment the already existing graph-optimization SLAM implementations. Section III succinctly formulates SLAM as a nonlinear least square estimation problem. Then, in the section IV we describe the characteristics of our new implementation. In the experimental evaluation section V we show the increased efficiency of our proposed scheme over the existing implementations. Conclusions and future work are given in Section VI.

II. Related Work

This work focuses on the implementation of nonlinear least square solvers, involving direct methods. Several successful implementations of graph optimization techniques for SLAM already exist and have been used in robotic applications. In general, they are based on similar algorithmic framework, repeatedly applying Cholesky or QR factorizations in an iterative Gauss-Newton or Levenberg-Marquardt nonlinear solver. g2o [7] is an easy to use, open-source implementation which has been proven to be very fast in batch mode. It exploits the sparse connectivity and operates on the block-structure of the underlying graph problem. A similar scheme was initially implemented in SSBA [8] and SPA [11] and it is based on block-oriented sparse matrix manipulation. Using blocks is a natural way to minimize cache misses, since the CPU can automatically pre-fetch the data as they are accessed. Nevertheless, taking care about the layout of the individual blocks in the memory is very important, otherwise the overhead of handling the blocks can easily outweigh the advantage of cache efficiency.

However, in SLAM the state changes every step when new observations need to be integrated into the system. For very large problems, updating and solving every step can become very expensive. Incremental smoothing and mapping (iSAM) allows efficiently solving a nonlinear graph optimization problem in every step [5]. The implementation incrementally updates the R factor obtained from the QR factorization and performs backsubstitution to find the solution. The sparsity of the R factor is ensured by periodic reorderings. Recently, the Bayes tree data-stucture [6], [13] was introduced to enable a better understanding of the link between sparse matrix factorization and inference in graphical models. The Bayes tree was applied to obtain iSAM2 [6], [13], which achieves high efficiency through incremental variable re-ordering and fluid relinearization, eliminating the need for periodic batch steps. When compared to the existing methods, iSAM2 performance finds a good balance between efficiency and accuracy. But still the complexity of maintaining the Bayes tree data structure can introduce several overheads.

The solutions proposed in this paper aim to improve the core of above-mentioned implementations, which is the block matrix manipulation and block matrix operations. Our scheme is general, and can be easily incorporated into advanced incremental algorithms such as iSAM. Even iSAM2, which relies on a tree-like data structure, can also benefit from the proposed scheme for the management of the dense blocks in memory.

III. SLAM as a Nonlinear Least Squares Problem

In robotics, SLAM is often formulated as a nonlinear least squares problem [3], which estimates a set of variables $\boldsymbol{\theta} = [\theta_1 \dots \theta_n]$ containing the robot trajectory and/or the position of landmarks in the environment, given a set of measurement constraints $\mathbf{z} = [z_1 \dots z_n]$. The goal is to obtain the maximum likelihood estimate (MLE) of a set of variables in $\boldsymbol{\theta}$ given the measurements in $\mathbf{z}$:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ P(\boldsymbol{\theta} \mid \mathbf{z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ \{-\log(P(\boldsymbol{\theta} \mid \mathbf{z})\}\ . \quad (1)$$

In SLAM, the measurement constraints involve rotations and are nonlinear. For every measurement $z_k = h_k(\theta_{i_k}, \theta_{j_k}) - v_k$ we assume the Gaussian distribution:

$$P(z_k \mid \theta_{i_k}, \theta_{j_k}) \propto \exp\left(-\frac{1}{2} \parallel h_k(\theta_{i_k}, \theta_{j_k}) - z_k \parallel^2_{\Sigma_k}\right), \quad (2)$$

where $h(\theta_{i_k}, \theta_{j_k})$ is the nonlinear measurement function, and where v_k is the normally distributed zero-mean noise with the covariance Σ_k. Finding the MLE from (1) is done by solving the following nonlinear least squares problem:

$$\boldsymbol{\theta}^* = \operatorname{argmin} \left\{ \sum_{k=1}^{m} \| h_k(\theta_{i_k}, \theta_{j_k}) - z_k \|^2_{\Sigma_k} \right\} , \quad (3)$$

where we minimize the sum of squared residual of the type:

$$r_k = h_k(\theta_{i_k}, \theta_{j_k}) - z_k \, . \quad (4)$$

Gathering all residuals in $\mathbf{r}(\boldsymbol{\theta}) = [r_1, \ldots, r_m]^\top$ and the measurement noise in $\mathbf{\Sigma} = diag([\Sigma_1, \ldots, \Sigma_m])$, the sum in (3) can be written in the vectorial form and expressed in terms of 2-norm:

$$\|\mathbf{r}(\boldsymbol{\theta})\|^2_{\mathbf{\Sigma}} = \mathbf{r}^\top(\boldsymbol{\theta})\, \mathbf{\Sigma}^{-1} \mathbf{r}(\boldsymbol{\theta}) = \left\| \mathbf{\Sigma}^{-\top\backslash 2} \mathbf{r}(\boldsymbol{\theta}) \right\|^2 . \quad (5)$$

Iterative methods such as Gauss-Newton or Levenberg-Marquardt are often used to solve the nonlinear problem in (3). They are based on a series of linear approximations of the nonlinear functions in $\mathbf{r}$ around the current linearization point $\boldsymbol{\theta}^i$:

$$\tilde{\mathbf{r}}(\boldsymbol{\theta}^i) = \mathbf{r}(\boldsymbol{\theta}^i) + J(\boldsymbol{\theta}^0)(\boldsymbol{\theta} - \boldsymbol{\theta}^i) \, , \quad (6)$$

where J is the Jacobian matrix which gathers the derivative of the components of $\mathbf{r}(\mathbf{x}_i)$. Gauss-Newton methods compute the correction $\boldsymbol{\delta} = (\boldsymbol{\theta} - \boldsymbol{\theta}^i)$ at each iteration. The linear least squares problem in $\boldsymbol{\delta}$ is:

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \frac{1}{2} \|A\, \boldsymbol{\delta} - \mathbf{b}\|^2 \, , \quad (7)$$

where the $A = \mathbf{\Sigma}^{-\top\backslash 2} J$ is the system matrix and $\mathbf{b} = -\mathbf{r}(\boldsymbol{\theta}^i)$ the right hand side (r.h.s.). For SLAM problems, the matrix A is in general sparse, but it can become very large when the robot performs long trajectories. The normalized system has the advantage of remaining of the size of the state even if the number of measurements increases:

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \frac{1}{2} \|\Lambda \boldsymbol{\delta} - \boldsymbol{\eta}\|^2 \, , \quad (8)$$

where $\Lambda = A^\top A$ is the information matrix and $\boldsymbol{\eta} = A^\top \mathbf{b}$ is the information vector.

A. Linear Solver

The linearized version of the problem introduced above can be efficiently solved using sparse direct optimization methods, either performing Cholesky or QR factorizations, followed by backsubstitution.

Cholesky factorization yields $\Lambda = R^\top R$, where $R^\top$ is the Cholesky factor, and a forward and back substitutions on $R^\top \mathbf{y} = A^\top \mathbf{b}$ and $R\, \boldsymbol{\delta} = \mathbf{y}$ first recovers $\mathbf{y}$, then the actual solution $\boldsymbol{\delta}$.

Alternatively, the normal equation in (8) can be skipped and *QR factorization* can be applied directly to the matrix A in (7), yielding $A = Q\, R$. The solution $\boldsymbol{\delta}$ can be directly obtained by backsubstitution in $R\boldsymbol{\delta} = \mathbf{d}$ where $\mathbf{d} = R^{-\top} A^\top \mathbf{b}$. Note, that Q is not explicitly formed; instead $\mathbf{b}$ is modified during factorization to obtain $\mathbf{d}$.

B. Incremental SLAM

Online robotic applications require fast and accurate methods for the estimation of the current position of the robot. In an online application, the state is *incremented* with a new robot position and/or a new landmark every step and it is *updated* with the corresponding measurements. This translates into adding new block columns to the matrix A corresponding to each new variable (pose/landmark) and new block rows corresponding to each measurement [3].

For the normalized equation in (8), the size of the matrix increments in rows and columns with the size of each new variable. Updating Λ and η is additive:

$$\tilde{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{21}^\top \\ \Lambda_{21} & \Lambda_{22} + \Omega \end{pmatrix} ; \quad \tilde{\boldsymbol{\eta}} = \begin{bmatrix} \boldsymbol{\eta}_1 \\ \boldsymbol{\eta}_2 + \boldsymbol{\omega} \end{bmatrix} , \quad (9)$$

where $\Omega = H^\top \Sigma_k^{-1} H$ and $\boldsymbol{\omega} = -H \Sigma_k^{-1\backslash 2} r_k$, H being the Jacobian of the measurement function.

For very large problems, updating and solving every step can become very expensive. Kaess et. al [5], [6], [13] proposed efficient algorithms to incrementally update and solve the linear systems. Those algorithmic improvements offer very good solutions to online SLAM but they are out of scope of this paper, which focuses on efficiently constructing the system at each iteration and speeding-up the basic arithmetic operations on sparse block matrices.

IV. IMPLEMENTATION DETAILS

In order to efficiently cope with very large nonlinear systems, the process of assembling and solving the sequence of linear systems must be as fast as possible. The data structure has to allow for both, efficiently re-computing the values of the matrices A or Λ and the r.h.s. $\mathbf{b}$ or $\boldsymbol{\eta}$ every time a new linearization point is available as well as efficiently updating the system when new measurements are available in incremental mode. One important characteristic of those matrices is their sparse block structure.

Operating on dense blocks is a natural way to support vectorization and improve cache efficiency without any additional effort. Also, the division of the data in blocks allows efficient data representation at their natural granularity, making it simple to reference the data inside the matrix and change their value when it's needed. In existing implementations, the blocks are usually allocated on the heap, and it can not be guaranteed that the blocks are allocated in close memory locations. If the blocks are allocated in distant memory locations, cache misses still occur. To improve that, our implementation uses segregated storage, which guarantees that the blocks are stored tightly next to each other. On the other hand, the arithmetic efficiency of block matrices is mostly reduced, compared to element-wise sparse matrices. That is because two extra loop counters for block rows and block columns are needed. Our implementation elegantly solves this issue using metaprogramming.

A. The data structure

In general, all existing block matrix schemes, including ours, involve the same data layout as CCS representation (or

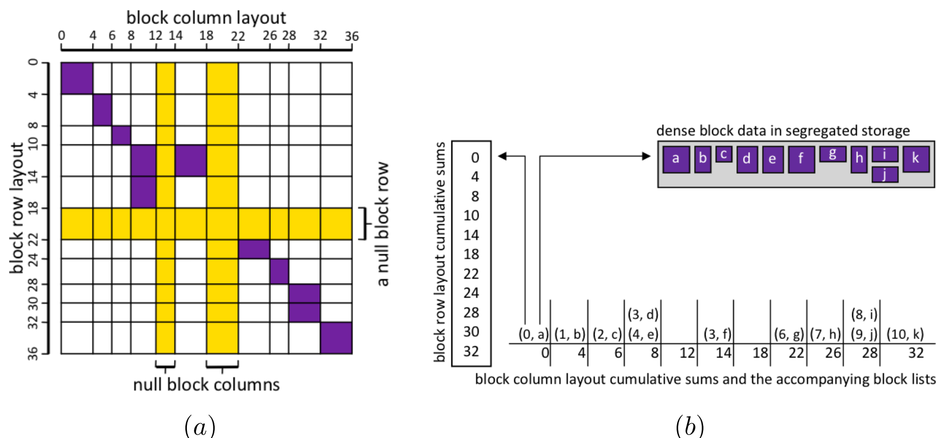


Fig. 2. Block row/column layout of a block matrix. a) An example of a sparse block matrix and the actual values of the cumulative block sum (on top and left side). Non-zero dense blocks are shown in violet. Yellow shows null rows/columns. b) Dense block data in segregate storage. On the bottom, we show the block column layout and the corresponding sorted list of pairs of type (iRL, pDB), where iRL is the index of the row layout, and pDB is the pointer to the block data in the memory.

equivalent), but use more complicated data structures such as trees or other higher abstract data structures to make the matrix structure editable. Trees, in particular, are used in the existing implementations such as g2o or SSBA.

In the proposed block matrix implementation, block row and block column layouts are described using the same structure, except that the columns also contain the non-zero matrix blocks. The structure is implemented as a sorted list of cumulative sums of block sizes (see Fig. 2 a)). The matrix blocks are also stored in a sorted list. Each matrix block contains row index and a pointer to matrix data. The data itself is allocated in forward-allocated segregated storage (see Fig. 2 b)), a storage model similar to a pool but only permitting allocation and de-allocation of elements from the end of the storage, in the same manner stacks do. This yields fast allocation and improves cache coherency.

In order to enable the unusually fast $O(1)$ block random access in arithmetic operations and also to facilitate error checking for incorrectly placed blocks, one important restriction on block and column layouts must be applied. The whole area of the matrix needs to be represented, which means that the layout of null rows or columns needs to be represented as well. Those are marked in yellow in Fig. 2 a) and their representation is shown in Fig. 2 b) where the fifth and sixth fields from the block column layout are empty.

This contrasts with the usual sparse block matrix representations, which only describe the layout of nonzero blocks without caring about the null elements in between. This comes at the cost of small increase in memory requirements, but only for the layout itself, not for the data. In general, for m block columns (or n rows), there can be maximum $floor(m/2)$ nonzero block columns (or $floor(n/2)$ nonzero block rows) that will require extra storage. Please note that for NLS problems there are no such null columns or rows, therefore, no extra space requirements apply.

B. Block matrix insertion

In order to write (scatter) a block in the matrix, the block column and block row needs to be resolved first. Adding a new block or column inside the matrix area takes $O(\log n)$ time. However, incrementally appending the matrix with blocks to or after the last row / column is a *constant time* operation, as it only needs to determine whether to create a new row / column at the end, or to use an existing one. This is a basic operation but frequently used in the context of incremental solvers. In the general case, in order to lookup a block, the position to insert the block is resolved in $O(\log n)+O(\log f)$ time, where f is the number of nonzero blocks in a column. Repeated block lookup is avoided in our implementation by storing a reference to the block after inserting it in the matrix. This is very useful for updating the A or Λ matrices every time a new linearization point is available. In this case, the new values of the blocks can be calculated directly inside the matrix, avoiding copying data or block lookup.

C. Arithmetic operations

The arithmetic operations on block matrices are carried out in the same manner as on element-wise sparse matrices, with the exception of handling matrix blocks instead of scalar values. Most of the arithmetic operations require block lookup at some point. In other existing block matrix implementations, the $O(\log n)$ lookup is used.

In our implementation, a mapping function between columns and rows of operand matrices is calculated, which enables direct access to the blocks in $O(1)$ time. The cost of calculating the mapping function is $O(n)$ in the number of block rows or block columns. Note that the mapping function needs to be only calculated once, before the arithmetic operation takes place. Also, the complexity involved is negligible, compared to the complexity of the arithmetic operation itself.

In SLAM and related problems, the possible block sizes correspond to DOF of the variables, and are known in

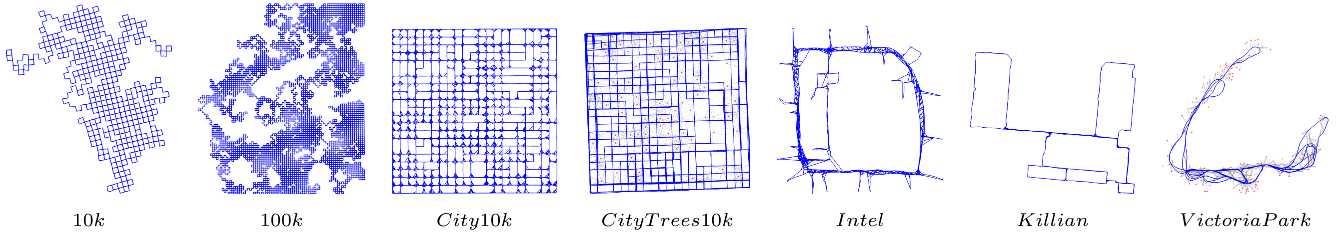


Fig. 3. The datasets used in our evaluations. Manhattan dataset is omitted due to space limitations.

TABLE I

TIME COMPARISONS OF THE BATCH SOLVERS (CH REFERS TO CHOLMOD AND CS REFERS TO CSPARSE).

	Manhattan	10k	100k	City10k	CityTrees10k	Intel	Killian
$g2o$ (CS)	0.0614	0.5539	10.8135	0.4855	0.1359	0.0066	0.0084
$g2o$ (CH)	0.0607	0.5497	9.41806	0.4491	0.1391	0.0070	0.0086
$iSAM$ (CS)	1.3641	2.9518	24.9582	1.4207	0.6245	0.0356	0.0535
$A-SLAM$ (CS)	0.0573	0.6341	10.4795	0.4635	0.1390	0.0126	0.0090
$A-SLAM$ (CH)	0.0613	0.6977	12.0097	0.5312	0.1469	0.0083	0.0095
Λ−SLAM (CS)	**0.0419**	**0.4852**	**9.2213**	**0.4203**	**0.0916**	**0.0052**	**0.0070**
$\Lambda-SLAM$ (CH)	0.0468	0.5798	11.0566	0.4563	0.1090	0.0060	0.0075
χ^2	6112.18	171545.45	8685.07	31931.41	548.50	559.05	0.000005
$iterations$	5	6	6	6	5	2	1

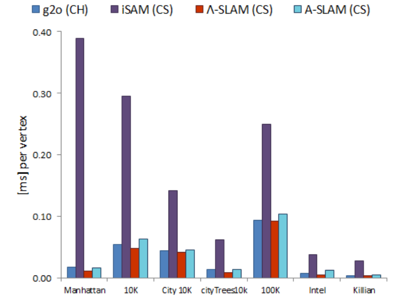
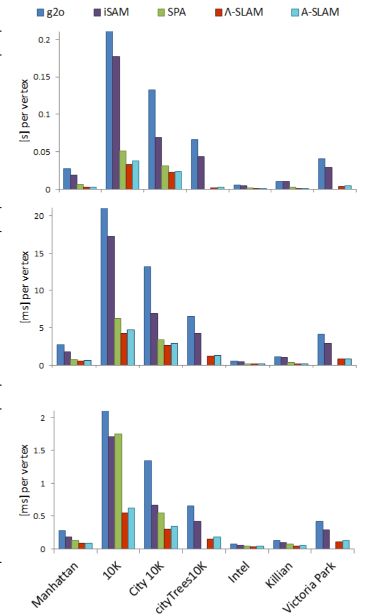


TABLE II

TIME COMPARISONS OF THE INCREMENTAL SOLVERS.

	Manhattan	10k	City10k	CityTrees10k	Intel	Killian	Victoria Park
Incremental-solve every step							
$g2o$	94.9096	2134.3000	1326.6600	659.1590	5.0513	20.8899	293.1010
$iSAM$	64.5844	1768.8400	693.7860	434.7500	4.4647	19.7519	209.1740
SPA	23.8834	515.2880	308.0680	N/A	1.4763	5.6260	N/A
$A-SLAM$	10.8883	377.7490	235.7910	25.2809	0.8829	2.4275	30.6333
Λ−SLAM	**10.0038**	**329.1840**	**222.5930**	**22.7070**	**0.8424**	**2.1485**	**28.0194**
Incremental-solve every 10 steps							
$g2o$	9.5326	211.2470	132.0070	65.0364	0.5245	2.1518	29.2946
$iSAM$	6.2510	172.8720	68.5533	42.7519	0.4541	1.9473	20.7089
SPA	2.5745	62.6485	33.4328	N/A	0.1689	0.6392	N/A
$A-SLAM$	2.1462	46.8314	28.8257	13.2880	0.1336	0.3194	6.0668
Λ−SLAM	**1.9560**	**42.0610**	**26.7019**	**12.0940**	**0.1227**	**0.2794**	**5.5461**
Incremental-solve every 100 steps							
$g2o$	0.9891	21.0767	13.3781	6.4883	0.0695	0.2443	2.9323
$iSAM$	0.6142	17.0565	6.6846	4.1876	0.0459	0.1915	2.0580
SPA	0.4446	17.4968	5.4739	N/A	0.0371	0.1426	N/A
$A-SLAM$	0.3059	6.2372	3.4363	1.8136	0.0339	0.0904	0.8963
Λ−SLAM	**0.2853**	**5.4294**	**3.0175**	**1.5028**	**0.0292**	**0.0845**	**0.7522**



advance. Therefore, our implementation employs advanced metaprogramming concepts to enable automatic compile-time generation of matrix operations code, containing a decision tree that chooses fixed-size function to perform operation on the block(s). This allows for loop unrolling and vectorization, making the block matrix operations faster than element-wise matrix operations. This is a very important advancement in the context of nonlinear solvers, as most of the existing implementations abandon arithmetic operations on block matrices altogether and resort to operations at the element level.

V. EXPERIMENTAL EVALUATION

In order to evaluate our new efficient block matrix scheme, we implemented two standard graph SLAM algorithms; one that builds the linear system in (7), which we call A-SLAM and another one that increments the information matrix in (8), which we call Λ-SLAM. We compared the timing results with similar state of the art implementations such as iSAM [5], g2o [7], and SPA [11] (a 2D SLAM variant of SSPA [8]). For SPA the svn revision 39478 of ROS (`http://www.ros.org/`) was used; for g2o, svn revision 29 from `http://openslam.org/` was used and for iSAM we used revision 7 from

https://svn.csail.mit.edu/isam.

We evaluate our implementation on five standard simulated datasets; *Manhattan* [14], *10k* and *100k* [4], *City10k* and *CityTree10k* [15] and three real datasets; *Intel* [16], *Killian Court* [17] and *Victoria park* (see Fig. 3). These are 2D SLAM datasets commonly used in evaluating graph-based SLAM implementations.

We performed all the tests on a computer with Intel Core $i5$ CPU 661 running at 3.33 GHz and 8 GB of RAM. This is a quad-core CPU without hyperthreading and with full SSE instruction set support. During the tests, the computer was not running any time-consuming processes in the background. Each test was run ten times and the average time was calculated in order to avoid measurement errors, especially on smaller datasets.

A. Other Implementations

All the implementations used for comparisons are based on similar algorithms, both in batch and incremental mode. Gauss-Newton non-linear solver was tested in all cases. iSAM has the possibility to provide incremental updates and solve every step and to perform expensive batch steps only when needed, but for comparison purposes we tested only the cases where batch, update and solve are done all together.

g2o and SPA use their own sparse block matrix implementation. In g2o, it is based on a dense vector of trees, where each tree contains blocks for one column. This allows relatively fast random access to matrix elements, only $O(\log f)$ compared to $O(\log f + \log n)$ in our implementation. However, our implementation always avoids accessing blocks randomly, while in g2o this complexity is enforced on block lookup in matrix operations, making them slower than both CSparse and our implementation. Overall, g2o is optimized for batch processing, but not for incremental solving. The good SPA timings come from the fact that their implementation is optimized for the specific 2D pose adjustment problem (or bundle adjustment problem in case on SSPA). In contrast, our implementation is general, allowing any combination of any block sizes.

In the case of iSAM, its sparse matrix storage does not use any block matrix scheme. A dense vector of sparse row vectors is used to represent a sparse matrix. This has some advantage for simple matrix manipulation, as each measurement modifies only certain sparse vectors, and it is also straightforward to perform matrix permutation. On the other hand, the storage is not cache friendly as data of each sparse vector are allocated separately.

B. Discussion of the results

Timing results for running batch and incremental SLAM are shown in tables IV-C and IV-C, respectively. Note that the accompanying figures show time per vertex, as it was hard to display the radically different times for all the datasets in a single plot. This makes the results comparable in scope of one dataset only, not between different datasets. *Victoria park* dataset is not included in the batch tests since it does not converge. Similarly, $100k$ dataset is too large to be executed incrementally, therefore we did not include it in the incremental tests. The last two rows of table IV-C reports values of the χ^2 and the number of iterations. Those are the same (or very close in the case of χ^2) for all the tested solvers. In incremental mode, the tests were done using the best linear solver from the batch mode (CHOLMOD - in the case of g2o and CSparse in the case of our implementation). The incremental results are split in three parts; solution calculated everytime, every 10 vertices and every 100 vertices.

Our implementation outperforms all the existing implementations in both batch and incremental mode. The comparison in batch mode shows a speed op of 10% when compared to the fastest implementation. This is mainly due to the proposed block matrix scheme, the algorithm being the same and the differences in the implementation style cannot cause large speedups.

However, observe that there is some imbalance between small speedup in batch mode and large speedup in incremental mode. This stems from the simple fact that in batch, the system is only constructed once and most of the time is spent in linear solver - CSparse [9] or CHOLMOD [10] - which we did not improve or modify in any way. For incremental mode, especially on large datasets, the efficient matrix operations start paying off, as there is a larger portion of time spent in updating the system.

Due to the efficient block matrix operations described in IV-C, the difference between A-SLAM and Λ-SLAM is very close to zero, as updating Λ as in (9) with all the measurements is just a parallel version of $A^\top A$ computation. Of course, when incrementing the system, the upper-left submatrix of Λ doesn't change and in Λ-SLAM, this computation is saved. In A-SLAM, $A^\top A$ must be calculated for the whole matrix, resulting in increased number of floating-point operations and slightly worse run times.

The difference between the use of CSparse [9] and CHOLMOD [10] is partly caused by different integer types being used in either library. We ran the tests on an Intel processor, where CSparse [9] is faster. Our development environment, however, was based on a pair of AMD processors, where CHOLMOD [10] is faster.

C. Block operations tests

Beyond the SLAM evaluation, we also ran matrix operations benchmarks on A and Λ matrices computed with the corresponding SLAM solution. Times for elementary sparse matrix operations, such as *compression*, *transpose*, *addition* and *multiplication* were measured. Performance of CSparse [9], g2o [7] and our implementation were compared. SPA [11] wasn't included because it's block matrix scheme is similar as in g2o. iSAM [5] wasn't included either, since it doesn't use any block matrix scheme. The results are shown in Fig. 4.

Observe that CSparse is very good with matrix compression, since it's data structure is the least complicated. But the compression must be performed every time the system is updated, making CSparse compression effectively

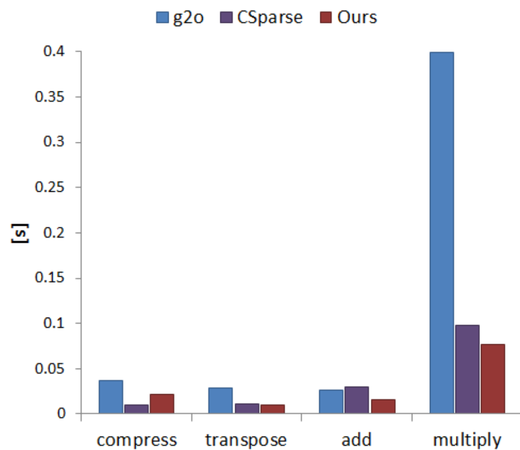


Fig. 4. Block matrix operations performance on SLAM dataset matrices.

slower after two iterations. In the other tests, our block matrix implementation outperforms CSparse. The most of the speedup comes from the use of vectorization. Furthermore, the block schemes prove to be more cache friendly than element-wise especially in the case of matrix transposition. In case of g2o [7], matrix transposition and multiplication is slower because of the use of the slow $O(\log n)$ block lookup, but they are not used in the optimization framework.

VI. Conclusions and future work

In this paper we evaluated a new implementation for block matrix manipulations, which are the core operations for many nonlinear least squares problems with applications in robotics. We targeted problems such as SLAM, which has a particular block structure, where the size of the blocks corresponds to the number of degrees of freedom of the variables.

The proposed scheme combines the advantages of *block-wise* schemes convenient in both, numeric and structural matrix modification and *element-wise*, which allows efficient arithmetic operation. The advantage of the new scheme was demonstrated through an exhaustive comparison with the existing implementations in SLAM, on several available datasets.

Even though the proposed scheme proved to significantly outperform the state-of-the-art implementations in incremental mode, several improvements from algorithmic point of view can be applied; incremental updates directly on the Cholesky factor, better ordering strategies (ordering is important to reduce the fill-in), changing only the blocks corresponding to affected variables.

The implementation itself can be improved. Some block matrix operations can be efficiently parralelized. In the current implementation, only the matrix vector product runs in parallel. Especially the time-consuming matrix multiplication needs to be parallelized, which should basically erase any differences between A-SLAM and Λ-SLAM in batch mode.

Finally, the block layout was designed with hardware acceleration in mind. This is very important for large scale problems, which can efficiently run on wide scale of accelerators, ranging from DSPs and FPGAs to clusters of GPUs.

VII. Acknowledgements

The research leading to these results has received funding from the European Union, 7th Framework Programme grants 316564-IMPART and 247772-SRS, Artemis JU grant 100233-R3-COP, and the IT4Innovations Centre of Excellence, grant n. CZ.1.05/1.1.00/02.0070, supported by Operational Programme Research and Development for Innovations funded by Structural Funds of the European Union and the state budget of the Czech Republic.

References

[1] C. Engels, H. Stewénius, and D. Nistér, “Bundle adjustment rules,” in *Symposium on Photogrammetric Computer Vision*, Sep 2006, pp. 266–271.
[2] F. Hecht, Y. J. Lee, J. R. Shewchuk, and J. F. O’Brien, “Updated sparse cholesky factors for corotational elastodynamics,” *ACM Transactions on Graphics*, vol. 31, no. 5, pp. 1–13, Oct. 2012, presented at SIGGRAPH 2012.
[3] F. Dellaert and M. Kaess, “Square Root SAM: Simultaneous localization and mapping via square root information smoothing,” *Intl. J. of Robotics Research*, vol. 25, no. 12, pp. 1181–1203, Dec 2006.
[4] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard, “A tree parameterization for efficiently computing maximum likelihood maps using gradient descent,” in *Robotics: Science and Systems (RSS)*, Jun 2007.
[5] M. Kaess, A. Ranganathan, and F. Dellaert, “iSAM: Incremental smoothing and mapping,” *IEEE Trans. Robotics*, vol. 24, no. 6, pp. 1365–1378, Dec 2008.
[6] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert, “iSAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.
[7] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, “g2o: A general framework for graph optimization,” in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.
[8] K. Konolige, “Sparse sparse bundle adjustment,” in *British Machine Vision Conference*, Aberystwyth, Wales, 08/2010 2010.
[9] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006.
[10] T. A. Davis and W. W. Hager, “Modifying a sparse cholesky factorization,” 1997.
[11] R. K. W. B. B. L. K. Konolige, G.Grisetti and R.Vincent, “Efficient sparse pose adjustment for 2d mapping,” in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2010, pp. 22–29.
[12] M. Kaess, V. Ila, R. Roberts, and F. Dellaert, “The Bayes tree: An algorithmic foundation for probabilistic robot mapping,” in *Intl. Workshop on the Algorithmic Foundations of Robotics*, Dec 2010.
[13] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, “iSAM2: Incremental smoothing and mapping using the Bayes tree,” *Intl. J. of Robotics Research*, vol. 31, pp. 217–236, Feb. 2012.
[14] E. Olson, “Robust and efficient robot mapping,” Ph.D. dissertation, Massachusetts Institute of Technology, 2008.
[15] M. Kaess, A. Ranganathan, and F. Dellaert, “iSAM: Fast incremental smoothing and mapping with efficient data association,” in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Rome, Italy, April 2007, pp. 1670–1677.
[16] A. Howard and N. Roy, “The robotics data set repository (Radish),” 2003. [Online]. Available: http://radish.sourceforge.net/
[17] M. Bosse, P. Newman, J. Leonard, and S. Teller, “Simultaneous localization and map building in large-scale cyclic environments using the Atlas framework,” *Intl. J. of Robotics Research*, vol. 23, no. 12, pp. 1113–1139, Dec 2004.