LinSol: An experimental linear solver library for C^{++}

Hao Lu, Lingaona Zhu, Sokratis Anagnostopoulos (team NaN), 02/02/20
(Github Repo: https://github.com/acse-2019/acse-5-assignment-nan)

Abstract

The following work presents the features and design of LinSol, a linear system solver library which attempts to experiment with a wide range of system types (dense, sparse and tridiagonal). The main objectives of the library development can be separated into four distinct categories: validation of results and functionality, evaluation of sparse vs dense methods, performance, and further research/testing involving comparison of different containers. At this point, it must be mentioned however, that although the performance of the examined methods was always considered throughout the development, this work rather aims at the understanding of such structures and not so at producing a competing library to the ones available. It also attempts to take advantage of as many capabilities of the C^{++} language as possible.

1. Validation of results

The validation of results was conducted using tests with different range of matrix sizes from 15 x 15 up to 1280 x 1280, using random matrix generator with different matrix types (i.e. dense, sparse and tridiagonal). A certain matrix was used to validate the potency of every solver before implementing into the timing process. The result showed consistency among all functions to the precision of 6 valid digits. Results were not compared using larger matrices as the result could be diversified due to precision limit of the laptop CPU.

2. Sparse vs Dense

This section compared the performance between sparse and dense matrices for two iterative methods (Jacobi and Gauss Seidel) and two direct methods (Cholesky and LU). According to the results, as the sparsity of input matrix increased, both Jacobi and Gauss Seidel methods for solving sparse matrices used less time to converge to the solution. As for Jacobi sparse (Fig. 1A), the converging speed appeared to always be faster than that Jacobi dense in each size of the matrix, which was the same as our expectation (Fig. 1A).

Gauss-Seidel method showed that it spent longer time to solve a sparse matrix than to solve a dense one when the sparsity was small (Fig. 1B). Also, as the sparsity increased, the converging speed of Gauss-Seidel method increased. In addition, for a sparsity of 0.9 (90%), the converging speed for sparse matrix exceeded that for dense matrix. The reason for the unexpected result was considered to be different error calculations (difference and 2-norm) of the output array in this iteration method. However, after unifying the way to calculate the error, the result still showed the same trend as before. Another reason for this was that the sparse matrix solver showed its advantage in matrix with large sparsity, while when non-zero numbers become larger, more time will be taken to allocate for nonzero value array and row indices.

LU decomposition behaved strangely during the timings when its answer is unapproachable for matrices larger than 400 x 400 (Fig. 1D). The speed is considerably slower than its dense counterpart. The possible reason is largely because the implementation of LU sparse solver is heavily nested with *for* loops and *if* statements, which would significantly impair its performance. The pattern showed in the graph also suggests the LU dense solver is insensitive to matrix sparsity.

Finally, during the Cholesky right-looking procedure a random sparse matrix is progressively filled with new elements, and as a result, the CPU time gain of the sparse handling technique becomes negligible (Fig. 1C). This deficiency is observed for all sparse matrices tested. However, according to Davis et al., 2016, when a sparse matrix is obtained from a real physical problem or application then the fill-in does not happen.

When solving dense matrices, each method had a stable slope (order of *O(3)* under increasing size of matrix and increasing sparsity). In addition, the computation time solvers took to solve dense matrices did not change a lot with different sparsity, which showed that solvers for dense matrices would not be influenced by sparsity.

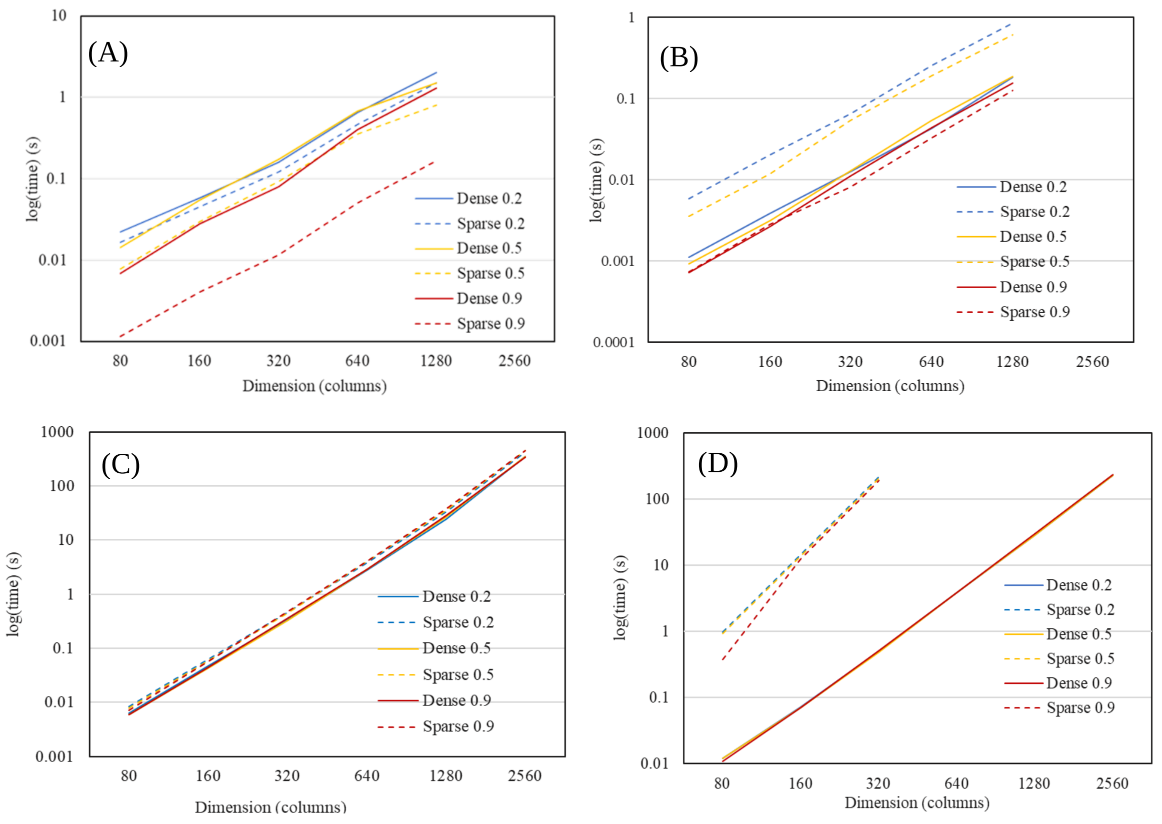

Figure 1. *Computation time vs dimensions for different sparsities; methods: (A) Jacobi, (B) Gauss-Seidel, (C) Cholesky, (D) LU.*

3. Performance

Approaches to optimise the code have been done as follows: implementing low level BLAS subroutines and vectorisation attempts (using __restrict in the code). Figure 2 shows the comparison between implementing default methods and after the implementation of BLAS routines. Out of the three solvers, Jacobi iterative solver shows expected overhead of the use of BLAS, whereas the other two methods almost overlapped. That could either be due to a very good written Gauss-Seidel algorithm or due to a not so good loop ordering, which doesn't allow for an improvement in the computation time with BLAS.

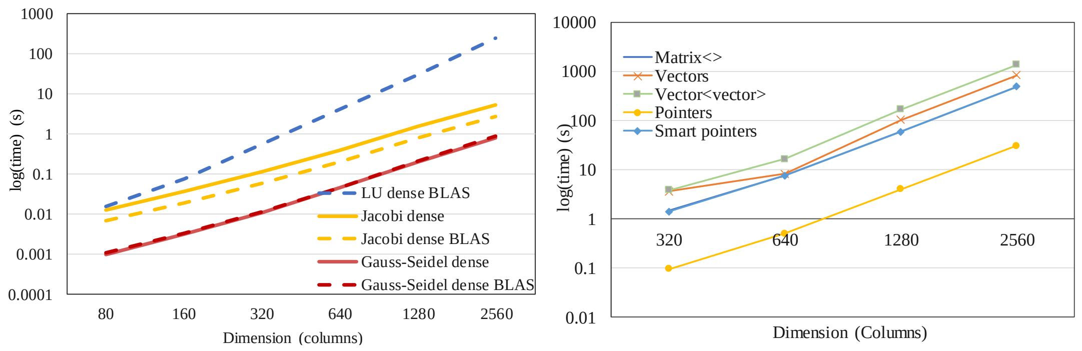

Figure 2. *(A): Computation time vs dimensions before and after using BLAS for several methods and (B) computation time vs dimensions of different containers used in Gauss elimination method.*

4. Further testing

As different types of containers can have a considerable effect on the performance of the algorithm, several different types were tested in order to evaluate their function. For this study, the Gaussian elimination solver was selected as the results should mainly be affected by the speed at which each is accessing the memory and not so on the solver method. The findings are presented in Figure 3, where again, the chosen dimensions of the system vary from 640 x 640 up to 2560 x 2560.

As expected, vector<vector> perform very poorly (almost 2x slower) compared to 1D vectors, since they are not contiguous in the memory (Fig. 2B). Smart pointers perform better than vectors and coincide with our Matrix<> template, which is expected since they were the main container used in the library.

Interestingly, it appears that raw pointers are on average two orders of magnitude (!) faster than any other container, as shown in Fig 2B. This could be attributed to the fact that they are the lowest level container and that provide additional capabilities due to smart pointers, such as deleting them exactly were instructed to avoid extra memory usage. However, such a difference in performance is extremely remarkable and thus further testing. It must be noted here that LinSol has mainly used smart pointers for their simplicity and convenience. Such a decision could be changed in future versions for performance increases. Finally, the increase in computation time is of *O(3)* for all the examined containers, which is expected for Gauss elimination.

Conclusion and future implementations

Future versions of the library could focus in significant performance increases by applying specific optimization techniques. Some of the current thoughts on this direction are:

a. Cache aware programming involves loop tiling and generally clever loop blocking in order to optimize nested loops in such a way that the data packets sent to the CPU can fit in the cache memory. An indicative attempt of this application is demonstrated in the Gauss-Seidel dense solver of the library, where the user can specify the number of row blocks to be calculated at a time. The machines that were used for this study so far had an average cache memory of 6Mb, which implies that the first step towards tiling could be to calculate no more than 800 x 800, double precision array elements at a time.
b. Parallelising chunks of array coefficient falls within the same scope as tiling, but also considering multi-threading usage. Clever array slicing and parallelising of such blocks can take advantage of the cache memory size but also of multiple CPU cores at the same time. This can mainly be applied in iterative methods that consist of independent steps like Jacobi.
c. Based on the findings of this study, raw pointers appear to drastically increase the performance of memory accessing at no cost, since their implementation (switching from smart pointer to raw) is trivial.
d. Enable optimizations like gcc -o3 compile commands and proper vectorization at an assembly level could also be implemented in future versions of the LinSol library.
e. Finally, the regular sparse matrices (banded, or block element matrices) could be further investigated, for two reasons: 1st, because such matrices are produced in many real applications' modelling, and 2nd, because the performance of some methods for sparse matrices can be much different when they are applied to regular than to irregular sparse matrices of same density and non-zero elements number. Therefore, the use of random sparse matrices may lead to misleading assessment of a solver behaviour and performance, as found for the Cholesky algorithm.

References

Davis T.A., Rajamanickam S. and Sid-Lakhdar W.M., "A survey of direct methods for sparse linear systems", *Technical Report, Department of Computer Science and Engineering*, 2016.