NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D3.4

Algorithm design for highly unsymmetric factorizations

April 2017

Document information

Scheduled delivery	2017-04-30
Actual delivery	2017-04-28
Version	1.0
Responsible partner	STFC

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2016-11-11	Iain Duff	Draft	0.1	Draft based on Word template produced by Lennart Edblom
2017-04-05	Iain Duff	Draft	0.2	Draft sent to internal reviewers
2017-04-27	Iain Duff	Final	1.0	Revised version after comments from internal reviewers

Author(s)

Iain Duff, STFC
Stojce Nakov, STFC

Internal reviewers

Jan Papez, INRIA
Mawussi Zounon, UNIMAN

Contributors

Tim Davis, Texas A & M University, USA

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for Deliverable D3.4:

> "*D3.4 Algorithm design for highly unsymmetric factorizations*
>
> Report on experimental algorithms for parallel Markowitz ordering, analysing various approaches and highlighting issues arising. Includes reporting on prototype code testing possible algorithms and solutions."

This deliverable is in the context of Task–3.3 (Direct methods for highly unsymmetric systems).

This deliverable discusses the design of algorithms for the factorization of highly unsymmetric matrices and is reporting on work in Task 3 of Workpackage 3. Our main work has been developing a parallel algorithm for the implementation of a Markowitz/threshold strategy. We first explain what we mean by "highly unsymmetric" matrices in Section 2 and present other attempts to design algorithms and software for this case. In Section 3, we discuss other considerations for this class of matrix factorizations including preprocessing techniques like scaling and partitioning emphasizing the current state of parallel algorithms for these phases. We then define what we mean by Markowitz/threshold in Section 4 before describing our parallel implementation in detail in Section 5. We give some preliminary results from running our prototype code in Section 6. Finally we provide some comments on future work in Section 7.

To fix our notation, we will assume that we are solving the system

$$Ax = b, \tag{1}$$

where A is a sparse matrix of dimensions $m \times n$. For the matrix A we only store coefficients that can be nonzero and call these entries. It is possible that some entries might have the numerical value zero either because of operations on them or because we are studying a set of matrices where an entry is sometimes nonzero but sometimes zero. This might happen, for example, if the matrix is the Jacobian of a nonlinear problem. An entry in row i and column j is designated by a_{ij}. The right-hand side vector b is of length m and the solution vector x is of length n. In this deliverable, we consider vectors x and b as dense. In our experiments for this deliverable, we only consider matrices with real entries although we will also develop versions of the code to handle matrices and vectors with complex entries. The methods that we use for solving Equation (1) are direct methods. That is we form an LDU factorization of a permutation of the matrix A, where L is a sparse lower triangular matrix, D is a diagonal matrix, and U is a sparse upper triangular matrix. The permutation is chosen to maintain sparsity in the matrices L and U while also producing a numerically stable factorization. We discuss how this is done in Section 4.

2 Highly unsymmetric matrices

We define a highly unsymmetric matrix as a matrix whose structure is not well approximated by the structure of $|A| + |A|^T$. Various authors have defined a measure of the asymmetry of a matrix and here we use that defined in [12] which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, viz.

$$si(A) = \frac{number_{i \neq j}\{a_{ij} * a_{ji} \neq 0\}}{nz\{A\}},$$

where si is called the symmetry index and $nz\{A\}$ is the number of off-diagonal entries in the matrix A. A symmetric matrix will thus have a symmetry index of 1.0. Matrices with symmetry indices of less than 0.5 can be considered highly unsymmetric and these are the main target of our current work.

There are many applications that give rise to such matrices, for example, econometric modelling, chemical engineering, and linear programming although the latter has developed a large cohort of software where special techniques are used to update the factors for sequences of very related matrices.

In contrast to the case of nearly symmetric matrices, there is little software for this class of matrices and almost no work on parallel algorithms. Available codes for this case include `MA48` and `HSL_MA48` [11], `UMFPACK` [6], `LUSOL` [14], and `KLU` [7].

3 Preprocessing highly unsymmetric matrices

An archetypical very highly unsymmetric matrix is a triangular matrix and a simple generalization of that form is a block triangular matrix (BTF). There are many algorithms that can find this form cheaply, and we will normally only consider matrices that are irreducible (that is cannot be permuted to a non-trivial BTF). There are good algorithms available to partition irreducible matrices to a bordered block diagonal form as shown in Figure 1. The market-leader is generally considered to be PaToH [2], but this code is not

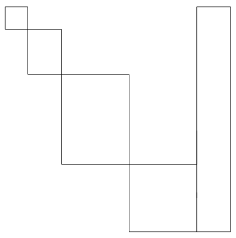


Figure 1: A singly bordered block diagonal form.

parallel so we recommend the PHG code of Zoltan [4] for doing this stage.

Numerical considerations are even stronger in the case of unsymmetric systems than in the solution of symmetric systems in Task 3.2, and it is crucial to scale the matrix prior to factorization. The market leader here is `MC64` [10] but again it is not parallel so we can use a simpler but still effective scaling like that given in [1].

4 Markowitz/threshold pivoting

We first note how important it is to order a sparse matrix before or during numerical factorization. To demonstrate this, we show a table from [9] where both the benefits of using sparsity and of ordering the matrix for factorization are clearly seen. There is a substantial reduction in both operations and storage for the factorization by using sparse data structures and another significant gain if a good ordering strategy is used.

Clearly for any pivot in Gaussian elimination the maximum fill-in[1] that can occur is the product of the number of other entries in the pivot row with the number of other

[1]An entry which is zero in A but is nonzero in the corresponding entry of the factors is termed fill-in.

Table 1: Operations for Gaussian elimination on `onetone1`, which has order 36 057 and 341 088 entries.

Treating matrix as	dense	sparse	
		unordered	ordered
Operations to factorize matrix ($\times 10^9$)	31 251	17.6	3.6
Storage for LU factors ($\times 10^6$)	1 300	33.4	5.3

entries in the pivot column. Thus if there are c_j entries in column j and r_i entries in row i, then we define the Markowitz cost for a potential pivot in row i, column j as

$$Mark_{ij} \; = \; (r_i - 1) \times (c_j - 1). \tag{2}$$

We choose candidate entries with low or minimum Markowitz count to reduce the amount of fill-in. Of course such a candidate would be unacceptable if its value was zero or very small relative to other entries. We therefore introduce a threshold of acceptability for a pivot and only consider entries a_{ij} that satisfy

$$|a_{ij}| \geq u * \max_k |a_{kj}|, \quad k = 1, \ldots, m \tag{3}$$

where u is a threshold parameter $0 < u \leq 1.0$. That is to say we only consider entries that are at least u times as large as the largest entry in modulus of all entries in the column. We call such entries eligible entries. If u were equal to 1.0 then we would be using partial pivoting that is the most common algorithm for dense matrices.

To continue with the factorization we must first update the remaining matrix using the outer product of the pivot row and column, updating the numerical entries and normally introducing fill-in. This is clearly a right-looking algorithm. For selecting the next pivot we then perform the Markowitz/threshold algorithm on this remaining updated matrix of order one less than the previous one, and we continue in this way until all $\min(m, n)$ pivots have been chosen. The algorithm is simple but the data structures to implement it efficiently, even in serial mode, are not. We consider the data structures that we use in detail in the next section.

5 Parallel implementation of Markowitz/threshold pivoting

For our parallel implementation, we essentially use the same algorithm, that is a threshold Markowitz/threshold algorithm using the same terminology as the previous section. In this implementation, we find a set of independent pivots that can be used in parallel and then use these as a block pivot to update in parallel the remaining matrix. We illustrate this in Figure 2. We then repeat these two steps on the updated Schur complement and continue doing this until either the Schur complement becomes denser than a preset value or the number of pivots found is less than a preset value. In fact, when we conducted the experiments that we describe in Section 6, we found that the number of pivots chosen at each stage was not, as might be expected, monotonically decreasing but the selection of a low number of pivots might be followed by a much larger number of independent pivots.

We thus monitor the number at each stage and do not switch unless the last few (the actual number is a parameter) steps have yielded only a very few (another parameter) pivots. We then switch to using a dense factorization routine on the remaining Schur complement. In our present implementation on multi-core machines we use `GETRF` from `PLASMA`[5].

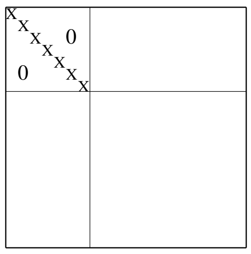


Figure 2: Block of independent pivots

When choosing the set of pivots in parallel, we note that the threshold test only needs information from one column and this is our first observation for obtaining significant parallelism. We thus launch our algorithm by scanning columns of the matrix independently. For each column we compute the largest entry in modulus and choose as a potential pivot in that column an entry that satisfies the threshold test, that is an entry at least u times the maximum just calculated, and is in a row with the least number of entries over all eligible entries in the column. It is possible that all eligible entries are in rows of high count so, although we still flag the entry as a possible pivot, we will not use it unless there are no other suitable pivots.

Having done this we then want to construct a set of independent pivots in parallel. We do this by a binary combination illustrated in Figure 3. We use a parameter to define

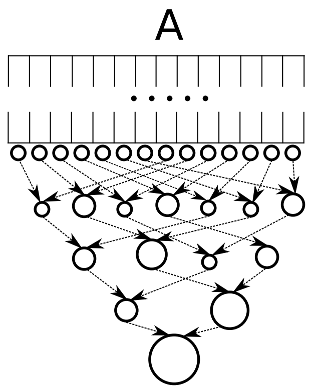


Figure 3: Combining pivots to get block pivot.

a block size and then select at random a set of columns of the matrix whose cardinality is the block size. We choose the columns at random because often the structure of the matrix militates against choosing consecutive columns. We then do a cheap scan of columns in the block to identify an independent set. Each block is independent and can be scanned in parallel. If we assume that the column we are seeking to combine with the current block pivot is j_1 and the block pivot is in the set of rows I_2 and columns J_2 and that the potential pivot in column j_1 is in row i_1 then the column is combined with the current block if there are no entries in positions (I_2, j_1) and (i_1, J_2). For checking whether a column yields an independent pivot we use an integer array of length n that flags whether a row has no entries in all the previously chosen columns. A similar flag is set for the columns so that the test comprises just two look ups followed by an update of the flags. This can be done without having to reset the flag array by incrementing flags at each step and only resetting if integer overflow occurs. We show experiments on the effect of the influence of the block size in the next section.

Having done this pass on all the blocks, we have sets of independent pivots of size up to the block size. We then combine these to get larger sets continuing to do so in a binary tree fashion as shown in Figure 3 until we have a single set of independent pivots.

In sequential codes like MA48[11], we select the eligible pivot which has the minimum Markowitz count, as defined in Equation (2). Because we want to get large blocks of independent pivots, we relax this by accepting eligible pivots within a factor of the minimum, that is an entry (i, j) can be chosen as a pivot if its Markowitz cost satisfies the condition

$$Mark_{ij} \leq \alpha_{Mark} \times Best_{Mark} \tag{4}$$

where the Markowitz factor α_{Mark} is greater than or equal to one and $Best_{Mark}$ is the lowest Markowitz count among all eligible entries.

Our next step is to perform all the pivot operations for this set of pivots in parallel. In effect what we have to do is a parallel sparse matrix by sparse matrix multiply to update the Schur complement.

Having done this, we then repeat the parallel pivot selection on the reduced matrix corresponding to the Schur complement that we have just computed.

We terminate the algorithm when either the last few steps of our algorithm ("few" is a parameter that we have set to 5 in our experiments) have failed to obtain a number of independent pivots greater than a preset threshold or the Schur complement reaches a preset density. At that stage, in the present code, we switch to using the PLASMA code GETRF for parallel dense LU factorization on the remaining Schur complement. We plan in later versions of the code to have a transitional stage where we use a parallel sparse direct code designed for relatively dense sparse matrices, such as the parallel LU factorization that will be developed in Workpackage 3.2.

Because of these various stages, we use four different data structures. The L and U factors continuously grow with the execution of our algorithm without changing the already computed part. For this reason, we use standard CSC and CSR storage for the L and U factors respectively. The values in the diagonal matrix D of the LDU factorization are stored in a separate array. On the other hand, the structure of the Schur complement changes because of the update operations. Additionally, at each step we must be able to determine if a set of pivots are mutually independent. For these reasons, we use a flexible CSC/CSR based structure. The numerical values are stored in the CSC fashion, while the CSR part stores only the nonzero structure of the matrix. In total, three large arrays are used, one index and one value array for the CSC part and one index array for the CSR part. Having the matrix structure stored by rows and columns provides an efficient way of updating the structures during pivot selection and Schur update. In order to cope with the dynamic nature of the Schur complement, within the CSR/CSC structure extra space is allocated at the end of each row or column. Each row and column is represented by an offset from the start of the corresponding array, by the number of entries each contains, with the amount of available free space. Additional memory is allocated at the end of each array which is managed by a garbage collector. For each block of free memory, the garbage collector stores the offset from the beginning of the array and its size. When fill-in to a row or a column consumes its available space, it is moved to the next available space provided by the garbage collector. Its old memory is marked as free memory and added to the garbage collector for future reuse. Similarly, space freed when a row or column becomes pivotal is likewise given to the garbage collector.

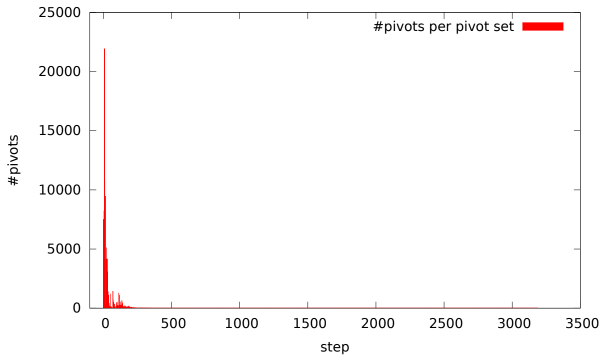
6 Preliminary results

In this section we present results of some experiments with our solver. The tests were performed on a multicore Haswell machine equipped with two Intel(R) Xeon(R) E5-2695 v3 CPUs with fourteen cores each (twenty eight cores in total). Each core is clocked at 2.3 GHz. All the results presented are sequential (mono-threaded execution). The main attributes of the matrices used in this study are given in Table 2. The matrix twotone is from the SparseSuite set of test matrices and the other two are from the Power Systems application supplied by Bernd Klöss of DigSILENT GmbH (see Deliverable 5.1).

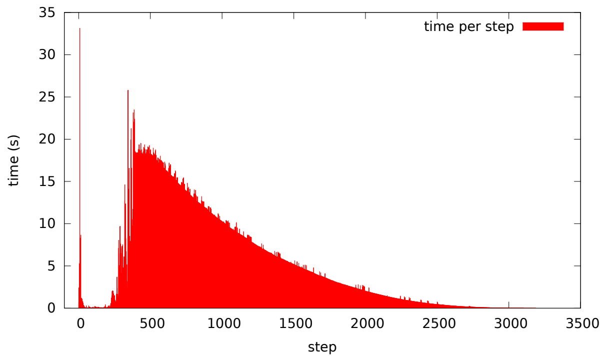
Table 2: Sizes of the matrices that are used in this study.

Matrix	n	nnz	si
twotone	120K	1.22M	0.26
LoadFlow_Newton_OuterLoop0_InnerLoop4_J	197K	3.70M	0.46
Jacobian_unbalancedLdf	203K	2.76M	0.80

As discussed in the previous section, at some point in our algorithm we switch to a dense solver. The reason for this is that once the Schur complement becomes too dense, we get only just a few pivots at a time and in addition the operations become more and more expensive. This is shown in Figure 4. At some point, we get sets of size one



(a)



(b)

Figure 4: The number of pivots (4a) and the time spent (4b) for each step when the matrix twotone is used.

only and the execution of each step takes a lot of time. This is considerably improved by switching to the dense solver (Figure 5). With this technique we are able to decrease the execution time from around 250 minutes to 93 seconds when the switch to the dense code is used.

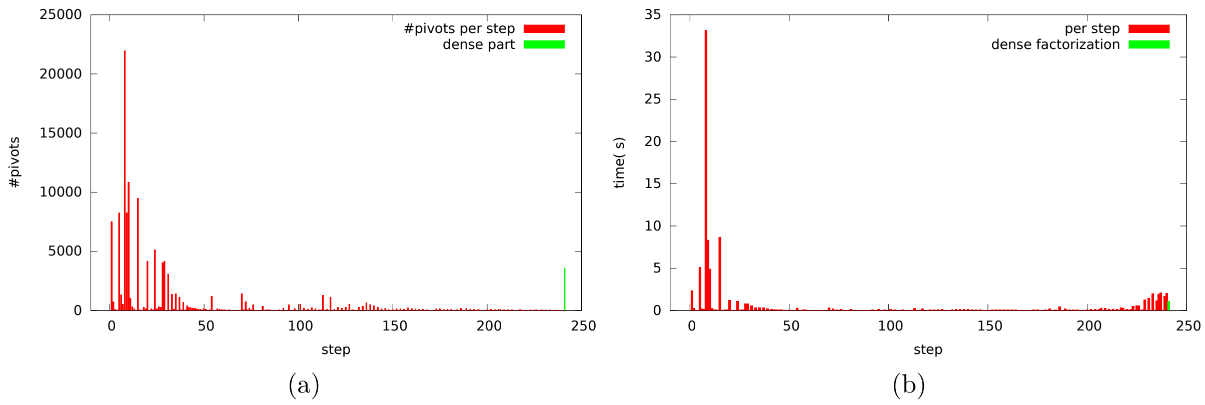


Figure 5: The number of pivots (5a) and the time spent (5b) for each step when the matrix twotone is used. Additionally, the number of pivots handled by the dense solver and the time spent in the dense solver are given at the end.

At the beginning of each step, we need to set up the initial sets of pivots that will be merged later on. The impact of the number of candidates per initial pivot set (the block size) on the average number of pivots found per step is given in Figure 6. We can see that for most of the different values of this parameter, our algorithm is able to create large enough sets. One interesting thing to notice is that when the LoadFlow_Newton_OuterLoop0_InnerLoop4_J matrix is used, the optimal value is 10. That points to the fact that the largest pivot sets are obtained when we start with a large number of small pivot sets. Since the merge is done using a binary tree and, at each level of the tree, all the merging can be done in parallel, it indicates that our algorithm could potentially be extremely parallel.

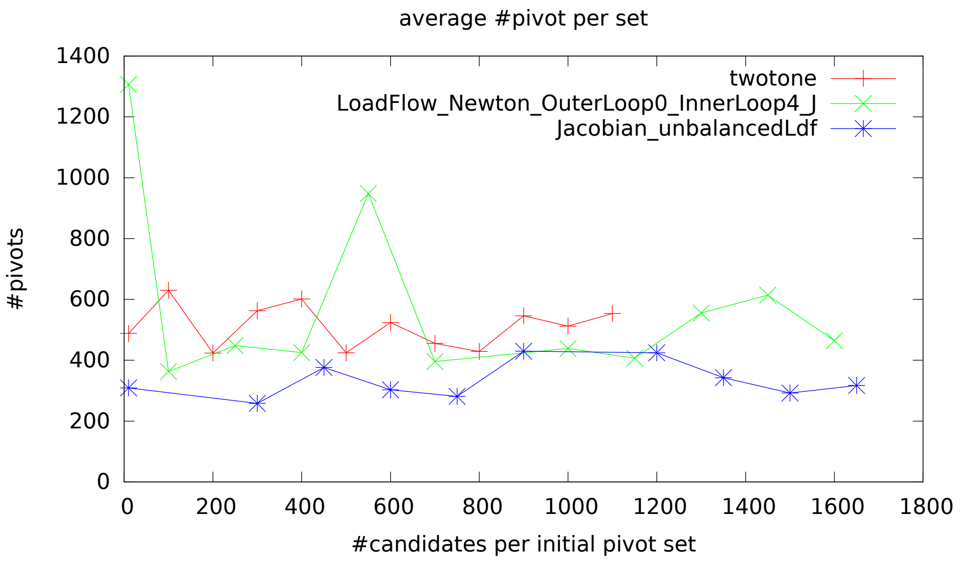


Figure 6: Impact of the number of candidates per initial pivot set on the average number of pivots per step.

Each pivot candidate must satisfy the Markowitz test in Equation (4). When we relax the constraint on the Markowitz cost, we accept pivots with higher Markowitz cost which will usually introduce more fill-in in the factors. The impact of the Markowitz factor α_{Mark} on the number of entries in the L and U factors is presented in Figure 7. In

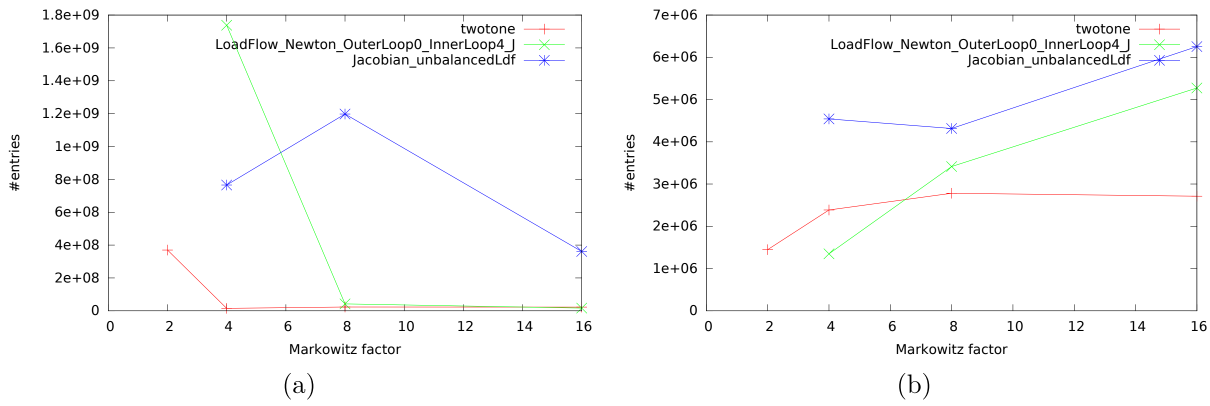


Figure 7: The impact of the Markowitz factor on the number of entries in the L and U factors. The total number of entries including the dense part of L and U is presented in 7a. Only the number of entries in the sparse part of L and U is shown in 7b.

Figure 7a the total number of entries in the L and U factors are shown and include the dense part of each of them. The expected behaviour would be that when the Markowitz factor is relaxed, the amount of fill-in increases, but our algorithm behaves in the opposite way. The reason for this is that for small values of the Markowitz factor, our algorithm is not able to find large enough sets of pivots so it switches to the dense solver earlier. On the other hand, if only the number of entries in the sparse part of the L and U factors is considered, then the number of entries increases with the relaxation of the Markowitz factor (see Figure 7b).

Another important parameter of our algorithm is the density threshold that forces the code to switch to a dense solver when the density of the Schur complement exceeds a certain threshold. The impact of this threshold is shown in Figure 8. The impact of the density threshold has similar behaviour to the Markowitz factor. When we constrain the density threshold, the switch to the dense code will happen earlier, resulting in a high number of entries for the whole L and U factors (see Figure 8a), but the number of entries in the sparse part of the factors is lower (see Figure 8b). That is, as the parameter is relaxed, the total number of entries in the factors decreases, but the number of entries in the sparse part increases.

In order to obtain a numerically correct solution, we have to take into account the numerical value of our pivots, see Equation (3). This will also impact the number of the entries in the factors. The effect of the threshold value is shown in Figure 9. If this parameter is relaxed (smaller values), then our algorithm is able to obtain a lower fill-in. The reason for this is that when smaller values for this threshold are used, our algorithm has a larger number of potential pivots at each step.

7 Conclusions and future work

The complexity of the algorithms and data structures for performing this parallel Markowitz search is very high and progress has been slower than originally planned. However, we now have a working, albeit embryonic, code and are in the process of tuning

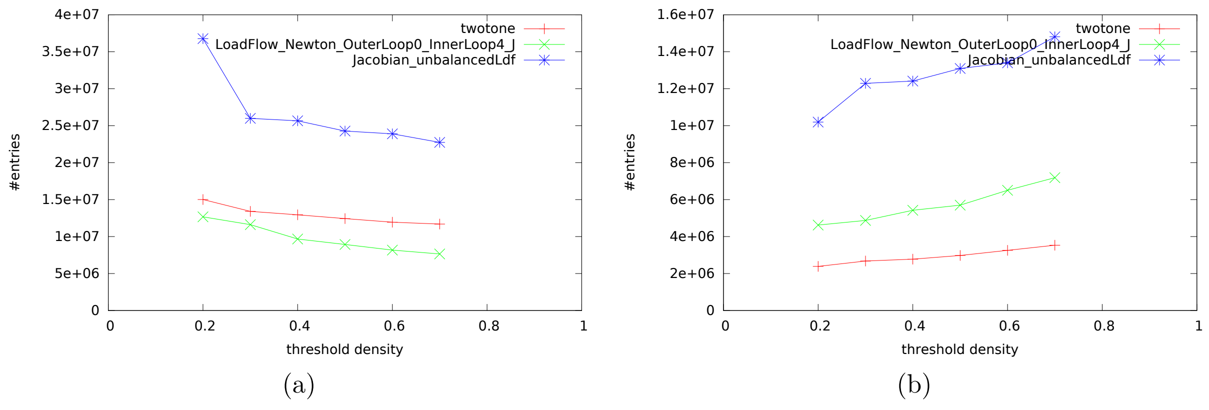


Figure 8: The impact of the density threshold on the number of entries in the L and U factors. The total number of entries including the dense part of L and U is presented in 8a. Only the number of entries in the sparse part of L and U is shown in 8b.

it and defining or reducing the number of parameters to have a user friendly release of a beta test version well before our M24 deadline for the release of our prototype code.

We are investigating current work on parallel sparse matrix by sparse matrix multiply, for example [3] and [8] to see if any of that work can help the update of our Schur complement update phase since, on some of our problems, we have found that this dominates the cost of pivot selection.

There are many instances when a sequence of sparse matrices must be factorized where each has the same sparsity structure and where the numerical values are similar so that the same pivot sequence can be used while maintaining a stable factorization. This might be the case when solving a nonlinear problem using Newton iterations. Clearly the complexity of parallel pivot selection and the use of sparse dynamic data structures can be avoided to yield a faster and more efficient factorization. This entry will be present in a later version of our code.

Although the major and more time consuming part of the solution of equation (1) lies in the LDU factorization, it is important to also provide efficient parallel code for the forward and back substitution using the triangular factors. We have already been investigating this in the context of GPUs [13] and plan to extend this work to use multi-core and heterogeneous machines also.

References

[1] P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar, *A parallel matrix scaling algorithm*, in High Performance Computing for Computational Science - VECPAR 2008: 8th International Conference, J. M. Palma, P. R. Amestoy, M. Daydé, M. Mattoso, and J. C. Lopes, eds., vol. 5336 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 301–313.

[2] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1860–1879.

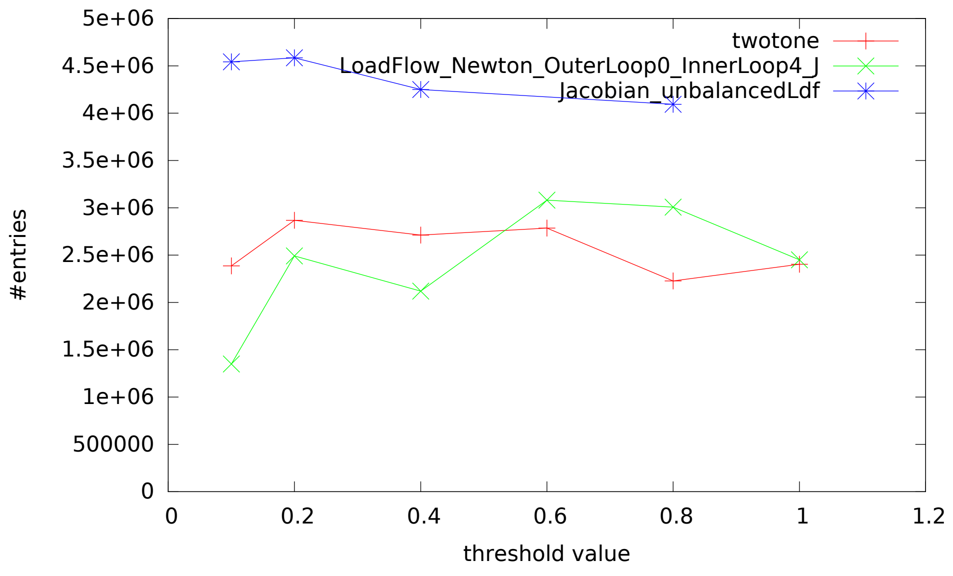


Figure 9: Impact of the value of the threshold parameter on the number of entries in the sparse part of L and U.

[3] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, *Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication*, SIAM J. Scientific Computing, 38 (2016), pp. C624–C651.

[4] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco, *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`.

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Computing, 35 (2009), pp. 38–53.

[6] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Analysis and Applications, 18 (1997), pp. 140–158.

[7] T. A. Davis and E. P. Natarajan, *Algorithm 907: KLU, A direct sparse solver for circuit simulation problems*, ACM Trans. Math. Softw., 37 (2010), p. 17 pages.

[8] M. Deveci, E. Boman, and S. Rajamanickam, *Performance portable sparse matrix-matrix multiplication for modern many-core architectures*, 2017. SIAM CSE'17 Conference, Atlanta, USA. February 2017.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices. Second Edition.*, Oxford University Press, Oxford, England, 2016.

[10] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Analysis and Applications, 22 (2001), pp. 973–996.

[11] I. S. Duff and J. K. Reid, *The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Softw., 22 (1996), pp. 187–226.

[12] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon, *Evaluation of orderings for unsymmetric sparse matrices*, SIAM Journal on Scientific and Statistical Computing, 7 (1987), pp. 600–624.

[13] W. Liu, A. Li, I. S. Duff, and B. Vinter, *A synchronization-free algorithm for parallel sparse triangular solves*, in Euro-Par 2016 LNCS 9833, P.-F. Dutot and D. Trystam, eds., Springer, Switzerland, 2016, pp. 1–14.

[14] M. A. Saunders, *LUSOL: Sparse LU for $Ax = b$*, 2009. `http://stanford.edu/group/SOL/lusol/`.