Issues in the Design of Scalable Out-of-Core Dense Symmetric Indefinite Factorization Algorithms

Peter E. Strazdins

Australian National University, Acton, ACT 0200. Australia

Peter.Strazdins@cs.anu.edu.au, <http://cs.anu.edu.au/~Peter.Strazdins>

Abstract. In the factorization of indefinite symmetric linear systems, symmetric pivoting is required to maintain numerical stability, while attaining a reduced floating point operation count.

However, symmetric pivoting presents many challenges in the design of efficient algorithms, and especially in the context of a parallel out-of-core solver for dense systems. Here, the search for a candidate pivot in order to eliminate a single column potentially requires a large number of messages and accesses of disk blocks.

In this paper, we look at the problems of scalability in terms of number of processors and the ratio of data size relative to aggregate memory capacity for these solvers. We find that diagonal pivoting methods which exploit locality of pivots offer the best potential to meet these demands. A left-looking algorithm based on an exhaustive block-search strategy for dense matrices is described and analysed; its scalability in terms of parallel I/O is dependent on being able to find stable pivots near or within the current elimination block.

1 Introduction

Large symmetric indefinite systems of equations arise in many applications, including incompressible flow computations, optimization of linear and non-linear programs, electromagnetic field analysis and data mining. An important special case, occurring in the latter two applications, are semi-definite and *weakly-indefinite* (i.e. most columns can be stably eliminated without pivoting) systems [1,2].

For the largest of such systems, even the aggregate memory of a distributed memory processor may not be sufficient [3]. Thus, out-of-core methods, scalable both with respect to matrix size and the number of processors, are needed in order to factor (and hence solve) these systems.

Algorithms for factoring $N \times N$ symmetric indefinite systems that are stable and yet exploit symmetry to have only $\frac{N^3}{3} + O(N^2)$ floating point operations are well known (see [4] and the references within). The diagonal pivoting methods are dominant in the literature, and several high-performance implementations of algorithms based on this method have been given [5,6,7], with some more recent papers considering distributed memory implementation [8,9].

Work so far on parallel out-of-core dense factorization algorithms has covered algorithms based on LU, LL^T and QR factorizations [10,11,12,13]. In all cases, left-looking versions of blocked algorithms are preferred; this is because these reduce the number of matrix elements written to disk from $O(N^3)$ to $O(N^2)$ [11] and permit easier checkpointing [13]. Within this, two approaches have been employed: a *slab-based* approach, where the block of columns currently being eliminated are kept in core, and a *block-based* approach, where only a part of the column block need be kept in core. The latter permits a wider column width to be used, which in turn permits greater re-use of data fetched from disk; thus it is more scalable with respect to linear system size relative to aggregate memory capacity [10,13]. However, it is not easily applied to LU factorization with partial (row) pivoting, as this requires a full column block to be in core [11].

While the difficulties of a block-based approach seem even greater for algorithms requiring symmetric pivoting (involving interchange of both row and columns), we shall see that diagonal pivoting methods offer a solution that can preserve stability guarantees.

This paper is organized as follows. Diagonal pivoting algorithms, with an outline of issues when applied to the parallel out-of-core case, are described in Section 2. Block- and slab-based algorithms based on locality of interchanges are described in Section 3, with some analysis of their potential performance. Conclusions are given in Section 4, together with a description of future work.

2 Diagonal Pivoting Algorithms

In the diagonal pivoting method, the decomposition $A = PLDL^TP^T$ is performed, where P is a permutation matrix, L is an $N \times N$ lower triangular matrix with a unit diagonal, and D is a block diagonal matrix with either 1×1 or 2×2 sub-blocks [4].

Figure 1 indicates the partial factorization of a left-looking blocked algorithm (see [8] for a description of the right-looking algorithm, and for more details on step 4). As A is symmetric, it is represented by a lower triangular matrix¹. Here, A_2 represents the current elimination block, and the first j_1 columns of L and

D (D_{11}) have already been factored and stored in $\begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix}$.

As this algorithm is left-looking, interchanges within A_2 resulting from the column eliminations in A_{11} are performed only when A_2 is ready for elimination (step 1).

In a left-looking LU factorization, it is possible to defer the row interchanges in A_1 (cf. step 5) until the factorization is complete; this can be useful as it

¹ Most implementations in the literature store a dense lower triangular matrix in a rectangular matrix, with memory locations corresponding to the upper triangular matrix being unused. With the advent of virtual memory, this incurs only an $O(N)$ storage overhead, as pages containing no elements of the lower matrix are never allocated.

1. apply pivots from A_{11} to $A_2 = \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix}$
2. $A_{22} \leftarrow A_{21}W_{21}^T$, $W_{21} = D_{11}A_{21}$
3. $A_{32} \leftarrow A_{31}W_{21}^T$
4. eliminate block column A_2 (applying pivots within A_2)
5. apply pivots (row interchanges) from A_2 to $A_1 = \begin{pmatrix} A_{21} \\ A_{31} \end{pmatrix}$

Fig. 1. Partial left-looking LDL^T factorization algorithm, showing a symmetric interchange between columns j and i

avoids having to repeatedly modify the already factored (and hence out-of-core) part of the matrix [11]. However, this is problematic when one tries to exploit symmetry as here the row interchanges operate on only part of the row and furthermore the current elimination block A_2 might only include one of the rows involved in the interchange.

The elimination of A_2 (step 4) proceeds column by column; in the elimination of column j , 3 cases can arise:

1. Eliminate using a 1×1 pivot from $a_{j,j}$. This corresponds to the definite case, and will be used when $a_{j,j}$ is sufficiently large, i.e. $|a_{j,j}| \geq \alpha\gamma_j$. Here $\gamma_j = \max(|a_{j+1:N,j}|)$ and $\alpha \in (0, 1)$ is a constant determining the growth rate (typically $\alpha = 0.5$).

2. Eliminate using a 1×1 pivot from $a_{i,i}$, where $i > j$. This corresponds to the semi-definite case; a symmetric interchange with row/columns i and j must be performed.
3. Eliminate using a 2×2 pivot using columns i' , $i' \geq j$, and i , $i \geq j + 1$ (this case produces a 2×2 sub-block at column j of D). This corresponds to the indefinite case; a symmetric interchange with rows/columns i', i and $j, j + 1$ is performed.

The tests used to decide between these cases, and the searches used to select column i (and i'), yield several algorithms based on the method, the most well-known being the several variants of the Bunch-Kaufman algorithm (where always $i' = j$ in Case 3). In all cases, the maxima of the candidate column i (i'), denoted γ_i ($\gamma_{i'}$), must be computed.

However, it has been recently shown for this algorithm there is no guarantee that the growth of L is bounded, and hence the algorithm has no overall strong stability guarantee [7]. Variants such as the *bounded Bunch-Kaufman* and *fast Bunch-Parlett* algorithms have been devised which overcome this problem [7].

2.1 Challenges for the Parallel Out-of-Core Case

In the parallel out-of-core case, the matrix A is distributed over a $P \times Q$ processor grid with an $r \times r$ block-cyclic matrix distribution, i.e. (storage) block (i, j) of A will be on processor $(i \bmod P, j \bmod Q)$. Assume that this distribution applies to both disk and memory; thus, loading a portion of A into memory involves only an access to the processor's local disk. Thus, the disk can be viewed simply as another level of the (distributed) memory hierarchy. Assume, for the moment, that the storage is column-oriented, both on disk and in memory.

In a left-looking blocked LDL^T factorization of A , as indicated in Figure 1, consider the elimination of column j where the candidate pivot i lies outside the current elimination block A_2 . For the sake of simplicity, assume a slab-based algorithm, i.e. all of the A_2 is in core.

To determine whether i is a suitable pivot, $a_i = (a_{i,j:i}, a_{i+1:N}, i)$ must be aligned with $a_j = a_{j:N,j}$, and, for a left-looking algorithm, *all* updates from the $j - 1$ eliminations so far must be applied to it. The alignment involves reading a potentially large number of disk blocks from remote processors, since parts of a row and a column must be accessed. The updates require all of A_1 to be read.

At this point, it may of course be discovered that i is not a suitable pivot and this process has to be repeated! In the out-of-core situation, it may be efficient to amortize this large cost by fetching a small block of pivots, and seeing if any are suitable.

If pivot i proves suitable, the interchange must then be completed. Firstly, a_i must be over-written by a_j (note that the portion in A_{33} must have the values of a_j *before* any updates were applied, for a left-looking algorithm). Secondly, at some later stage, the interchanges of rows i and j in A_1 need to be performed (step 5). However, with a slab-based algorithm, these can be performed in-core whenever A_1 is loaded into memory [11].

Thus, if a significant number of searches for pivots from outside the current elimination block must be made, algorithm performance will be severely degraded. From the point of view of having to update a_i , it has been argued right-looking algorithms are strongly favoured for LDL^T [14], and may even be preferable for the out-of-core case.

However, if i is within the current elimination block, the only overhead will be in the actual message exchanges, and even this will only occur if column i is outside the storage block of column j [8].

2.2 Elimination Block Search Strategies

A method searching for (stable) pivots in the current elimination block used for sparse matrices is the *exhaustive pivoting strategy* [7], which originates from the earlier MA27 codes by Duff and Reid [15]. This technique is used in multi-frontal methods, where this has a large payoff in preserving the sparsity of the matrices [7].

Here, a block of ω_a columns currently under consideration for elimination are arranged in a queue. Each iteration of the search selects column i (initially from the head of the queue, working inwards). After computing γ_i , it is then matched with each column i' preceding it in the queue. Note that at this point, it must be ensured that all updates from previous eliminations have been applied to columns i and i' .

A simple test involving matrix elements at positions (i, i) , (i, i') and (i', i') , and γ_i and $\gamma_{i'}$ determines whether columns i and i' can form a stable 2×2 pivot [7]. If it fails, column i is then considered for a 1×1 pivot.

One remaining point is what to do if no stable pivots are found in the current block. A number of fresh columns (from the trailing sub-matrix A_{33}) will be then brought to the head of the queue, and the whole process will then be repeated. While the queue size ω_a may thus grow to be $O(N)$, eventually the whole factorization will complete [7].

This strategy has been adapted for the dense parallel case [9]. The elimination block is the current storage block and so is of fixed size; thus all columns (rows) of the block are held within a column (row) of the processor grid (cf. Figure 1). Thus interchanges within the block can be performed without communication. If no stable pivot from within the block could be found for a column, it is eliminated using a pivot from outside using the bounded Bunch-Kaufman algorithm. Here the storage block size is determined by optimal matrix-matrix multiply (cf. steps 2–3 in Figure 1) performance, which is in turn determined by the cache size.

Such a more rigid scheme is a good tradeoff in the dense parallel (in-core) case: it permits the BLAS computational routines to operate a peak efficiency, whereas the cost for searching for a pivot outside the elimination block is low enough to be tolerated provided it occurs in the minority of cases. Indeed, it was found that best performance was obtained by limiting the search to $\omega_s \leq \omega_a$ columns in advance (e.g. $\omega_s = 16$ with $r = \omega_a = 64$ being optimal on a Fujitsu AP3000) [9], as the overhead of computing a number of column maxima will at some point exceed the communication costs of a symmetric interchange.

With this limited strategy, it was found that in the worst case (for highly indefinite matrices) only $0.15N$ searches outside the elimination block were required. This number reduced to $< 0.05N$ for the ‘weakly indefinite’ matrices typically generated to electromagnetic compatibility analysis [9].

However, as we have seen in Section 2.1 for the out-of-core situation, the cost of searching outside the elimination block is much higher, and more exhaustive strategies will be favored.

3 Diagonal Pivoting Methods for a Parallel Out-of-Core Algorithm

In this section, we develop both slab- and block- based algorithms exploiting locality of interchanges, and give some analysis of their behaviors.

3.1 Blocking and Data Layout Issues

To obtain high performance, such an algorithm must consider all levels of the (parallel) memory hierarchy, and exploit locality in searching for pivots as much as possible. Correspondingly, the following blocking factors can be identified.

- ω_a , the blocking factor of the outer algorithm ($\omega_a = j_2 - j_1 + 1$ in Figure 1). This should be as large as possible to maximize data-reuse at the disk level. Let M denote the amount memory available for the algorithm; for a slab-based algorithm, ω_a is constrained by:

$$\frac{N\omega_a}{PQ} \leq M \quad (1)$$

- ω_c , the optimal blocking factor for matrix-matrix multiply (as performed in step 3 in Figure 1), determined by maximizing data-reuse at the cache level. An optimal algorithm will require $\omega_a \geq \omega_c$; with $\omega_c < \omega_a$, two levels of blocking may be incorporated [11].
- r , the storage block size, primarily determined by concerns of minimizing communication latencies [14]. The situations $r = \omega_c$ (storage blocking) and $r \approx 1$ (algorithmic blocking) are typical [14]. Note that to fully utilize aggregate memory capacity, we will require:

$$\omega_a \geq \max(P, Q)r \quad (2)$$

- ω_s , the number of columns to be searched in the elimination block search strategy (see Section 2.2); this should be sufficiently large to ensure a successful search most of the time. However, as all of the columns must be kept up-to-date after each elimination, so that their maxima can then be computed, all of these columns must be in core, i.e.:

$$\frac{N\omega_s}{PQ} \leq M \quad (3)$$

There may be a reduction in communication costs if $\omega_s \leq r$ holds [8].

- ω_d , where ω_d^2 equals the disk block size. For a block-based algorithm, $\omega_a^2 \gg \omega_d^2$ will be required to amortize disk latencies.

A second issue is the organization of A upon disk: while in Section 2.1 we assumed it conformed to the block-cyclic data distribution, there is still freedom over whether disk blocks are column or row-oriented, or oriented towards block rows and columns (with ω_s and r being candidates for the sizes of these blocks). The choice depends on the dominant access patterns of the algorithm.

3.2 A Slab-Based Algorithm Exploiting Locality in Interchanges

We will now describe how the partial left-looking factorization algorithm of Figure 1 can be combined with an exhaustive elimination block search strategy, as described in Section 2.2, for a slab-based algorithm using a fixed width of ω_a .

Assume that there are u_1 uneliminated columns left over from the previous stage, where $0 \leq u_1 < \omega_a$, which are still in core. These columns will become the first u_1 columns of A_2 for this stage of the algorithm. The last $\omega_a - u_1$ columns of A_2 are now loaded into core, and steps 1–3 in Figure 1 are applied to only these columns.

Insert a new step 3a: interchange the first u'_1 columns of A_2 with the last u'_1 columns, where $u'_1 = \min(u_1, \omega_a - u_1)$. The motivation for this is to move the ‘difficult to eliminate’ columns to the back of A_2 , so that updates from the fresh columns have a chance of changing these columns.

Step 4 can use an exhaustive block search strategy ($\omega_s = \omega_a$); after completion, the $\omega_a - u_2$, $0 \leq u_2 < \omega_a$, successfully eliminated columns from this stage are written back to disk. To ensure progress, if the elimination block search strategy resulted in no eliminations, at least one column must be eliminated using non-local interchanges, e.g. by the bounded Bunch-Kaufman method.

Thus, this algorithm proceeds operating on a series of overlapping slabs; provided the elimination block search strategy always can eliminate one column, all interchanges will be within these slabs. In this case, no interchanges could have yet been applied to columns $j_1 + u_1 : N$ from previous stages, so step 1 is empty and A_{21} and A_{31} will have their rows aligned with A_2 for the updates of steps 2–3. Furthermore, step 5 will not require any updates to A_{31} .

Note that A_{21} and A_{31} can be read in column blocks of width much less than ω_a (but the width should be at least ω_c for good matrix multiply performance), which are then applied to A_2 ; provided $\omega_c \ll \omega_a$, only a negligible amount of extra memory need be reserved for them [12,13].

The performance of this algorithm thus depends on the u_i ’s being small, e.g. $u_i \leq 0.2\omega_a$, i.e. the (exhaustive) block search strategy is highly successful at finding stable pivots. The pivoting behavior of the limited strategy of [9] mentioned in Section 2.2 provide encouragement that this will hold in practice.

In this algorithm, ω_a is limited by Equation 1. Achieving good load balance in terms of both computation and memory utilization (Equation 2) favours a low value of r . This, plus the fact that the slabs can overlap, means that the elimination blocks cannot be confined within storage blocks. In other words, an

algorithmic blocking regime must be used here, even though this means interchanges within a slab will require communication.

Assuming $u_i \approx 0$, the number of matrix elements read from disk are dominated by steps 2–3:

$$\sum_{i=0}^{N/\omega_a} i\omega_a(N - i\omega_a) = \frac{N^3}{6\omega_a} + O(N^2) \quad (4)$$

and approximately $2 \cdot \frac{N^2}{2}$ matrix elements are written (including step 5).

As the dominant disk accesses are column oriented, organizing disk blocks in a column-based fashion is desirable. However, placing $r' > 1$ columns per disk block will reduce the number of blocks read in step 5.

3.3 The Need for Improved Memory Scalability

Considering Equation 4 and 1, the question arises of whether, on contemporary parallel machines, a slab-based algorithm is sufficient for good performance for problem sizes that can be solved in ‘reasonable time’.

For a processor with 0.5 GB (equivalent to 6.7×10^7 double precision words) of memory available to matrix data and can sustain 1 GFLOPS over a factorization, a matrix of size $N_1 = 10^4$ will (just) fit in core, and take $t_1 \approx 5$ min to factorize. For a matrix of size $N_2 = 10^5$, a slab of width $\omega_a \approx 670$ would fit within core and is likely to be wide enough for high re-use of disk accesses; this computation would require $(\frac{N_2}{N_1})^3 t_1 = 10t_1$ time, somewhat under 100 hours.

For a $P \times P$ grid of these processors, where $P = 10$, a matrix of size $N_2 = 10^5$ would just fit in core and, assuming an ideal speedup, factorize in time $(\frac{N_2}{N_1})^3 / P^2 t_1 = 10t_1$. For a matrix of size $N_3 = 10^6$, a slab of (ample) width $\omega_a \approx 660 \times 10$ would fit in core, and would factor in time of $(\frac{N_3}{N_1})^3 / P^2 t_1 = 10^4 t_1$ (over one month).

Thus, unless one wishes to factor systems requiring much more time than this, a slab-based algorithm will be adequate for moderate-sized grids of contemporary machines.

3.4 A Block-Based Algorithm

The algorithm of Section 3.2 can be converted to require a slab of width of only $\omega_s < \omega_a$, where instead ω_s is constrained by Equation 3. As indicated in Section 3.3, the resulting value of ω_s should be ample on contemporary parallel machines to permit a successful block search strategy and good load balance.

The idea is to simply to apply the left-looking factorization algorithm internally to step 4 with a blocking factor of ω_s . $\omega_s \geq \omega_c$ is desirable for good cache performance (and, as we have seen in Section 3.3, likely to be obtainable for situations of interest).

This implies that columns of A_2 must be read repeatedly as they were for A_1 in the slab algorithm, with the total number of such elements being given by:

$$\sum_{i=0}^{N/\omega_a} \sum_{j=0}^{\omega_a/\omega_s} i\omega_a \cdot j\omega_s = \frac{N^2 \omega_a}{4\omega_s} + O(N) \quad (5)$$

which are dominated by the number of reads in Equation 4.

Steps 2–3 can proceed with a block-based algorithm as is done for LL^T in [13], using $\omega_a \times \omega_a$ size blocks, and similarly for step 5. As these account for the dominant number of disk reads, storing the factored columns L in a row-block oriented fashion is desirable.

4 Conclusions and Future Work

Factorizing and hence solving symmetric indefinite systems is an interesting computation where there is a tradeoff in accuracy and performance.

Especially for the parallel out-of-core case, we have shown that there are potentially very large overheads in terms of disk accesses and messages in this computation, due with (long-distance) symmetric interchanges.

We have developed both slab and a (limited)-based algorithms which avoid this by having overlapping elimination blocks and search exhaustively for stable pivots within each block. The flexibility permitted in pivot searches by the diagonal pivoting methods was an important ingredient.

In either case, balanced memory utilization is important, favoring a small storage block size (i.e. the algorithmic blocking technique), even if this results in some extra communications.

However, the algorithms' performance is highly dependent on finding stable pivots within (the next ω_s columns of) the current elimination block in the majority of cases.

While the block-based algorithm has a limited scalability in that the next ω_s columns must be kept in core, this seems not to be crucial given the memory capacity of contemporary processors. Indeed, it is unclear that for medium-to-large sized contemporary parallel processors whether the greater scalability of the block-based algorithms is really necessary for problems that can be solved in reasonable time, given their extra complexity in implementation.

Future work will firstly involve investigating the success of block search strategy for large in-core algorithms. If proved successful, as expected from earlier studies, the out-of-core algorithms will be implemented. Their performance with respect to disk and communication overheads need then to be evaluated, and the latter also compared with the performance in the in-core case with existing implementations [8,9].

References

1. Homma, K., Nagase, K., Noro, M., Strazdins, P., Yamagajo, T.: Frequency Interpolation Methods for Accelerating Parallel EMC Analysis. In: Proceedings of the 15th International Parallel and Distributed Processing Symposium (for the 2nd Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications), San Francisco, IEEE Press (2001) 10
2. Christen, P., Hegland, M., Nielsen, O., Roberts, S., Strazdins, P., Altas, I.: Scalable Parallel Algorithms for Surface Fitting and Data Mining. *Parallel Computing* **27** (2001) 941–961

3. Cwik, T., Patterson, J., Scott, D.: Electromagnetic Scattering Calculations on the Intel Touchstone Delta. In: Proceedings of the Supercomputing 92, IEEE Press (1992) 538–542
4. Golub, G., Loan, C.V.: Matrix Computations. second edn. John Hopkins University Press, Baltimore (1989)
5. Anderson, C., Dongarra, J.: Evaluating Block Algorithm Variants in LAPACK. In: Fourth SIAM Conference for Parallel Processing for Scientific Computing, Chicago (1989) 6 pages.
6. Jones, M.T., Patrick, M.L.: Factoring Symmetric Indefinite Matrices on High-Performance Architectures. *SIAM Journal on Matrix Analysis and Applications* **12** (1991) 273–283
7. Ashcraft, C., Grimes, R.G., Lewis, J.G.: Accurate Symmetric Indefinite Linear Equation Solvers. *SIMAX* **20** (1998) 513–561
8. Strazdins, P.E.: Accelerated methods for performing the LDLT decomposition. *ANZIAM* **42** (2000) C1328–C1355
9. Strazdins, P., Lewis, J.: An Efficient and Stable Method for Parallel Factorization of Symmetric Indefinite Matrices. In: Proceedings of The 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region, Gold Coast (2001) 13
10. Toledo, S., Gustavson, F.: The design and implementation of solar, a portable library for scalable out-of-core linear algebra computations. In: Proceedings of the 4th Annual Workshop on I/O in Parallel and Distributed Systems, Philadelphia (1996) 28–40
11. Dongarra, J.J., Hammarling, S., Walker, D.W.: Key Concepts for Parallel Out-of-Core LU Factorizations. Technical Report CS-96-324, LAPACK Working Note 110, Computer Science Dept, University of Tennessee, Knoxville (1996)
12. D’Azevedo, E.F., Dongarra, J.J.: Key Concepts for Parallel Out-of-Core LU Factorizations. Technical Report CS-97-347, LAPACK Working Note 118, Computer Science Dept, University of Tennessee, Knoxville (1997)
13. Gunter, B.C., Riley, W., van de Geijn, R.A.: Parallel Out-of-core Cholesky and QR Factorizations with POOCLAPACK. In: Proceedings of the 15th International Parallel and Distributed Processing Symposium (for the 2nd Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications), San Francisco, IEEE Press (2001) 10
14. Strazdins, P.E.: Parallelizing Dense Symmetric Indefinite Solvers. In: PART’99: The 6th Annual Australasian Conference on Parallel And Real-Time Systems, Melbourne, Springer-Verlag (1999) 398–410
15. Duff, I., Reid, J.: MA27: A Set of Fortran Subroutines for Solving Sparse Symmetric Sets of Linear Equations. Technical Report AERE R 10533, Harwell (1983)