A Comparison between Three External Memory Algorithms for Factoring Sparse Matrices

F. Dobrian and A. Pothen

1 Introduction

We investigate a problem that arises in designing external-storage algorithms for solving large, sparse systems of linear equations ($Ax = b$) by direct (factorization-based) methods. We consider the Cholesky factorization, $A = LL^T$, of symmetric positive definite systems. Throughout this paper we assume that the coefficient matrix is already ordered by a fill reducing ordering algorithm.

The factorization is known to require a significant amount of storage, mainly because L is usually much larger than A. As a consequence, it may not be possible to factor a large matrix *in-core*, using only the internal storage (the core). In this case it is necessary to perform an *out-of-core* factorization, using the external storage as well [1, 11]. The movement of data between the internal and the external storage represents the *external traffic* (or I/O traffic).

An out-of-core computation requires a core manager. This may have to reorganize the in-core data during the computation, determining an *internal traffic* as well.

In this paper we discuss a particular case in which only the factor entries are stored externally and minimum I/O is performed. The external traffic is minimized if each factor entry is written exactly once and never read [4, 5, 6].

This research is motivated by the need to solve large sparse systems of equations. While it is possible to use the implicit I/O performed by the operating system's virtual memory mechanism, our experience shows that implicit I/O significantly degrades performance.

The particular case we consider in this paper is part of the larger picture. A general out-of-core factorization must be allowed to perform both read and write operations, perhaps for both A and L. However, it is possible to determine domains on which the type of factorization we currently focus on can be used [10]. As a consequence, the overall I/O is reduced.

We consider all three major column-oriented factorization algorithms: *left-looking*, *right-looking* and *multifrontal* [8, 9]. We compare them using two properties: the minimum size of the core that allows a minimum I/O out-of-core factorization and the amount of internal traffic. We report results for 2d and 3d model problems.

Various theoretical aspects of the out-of-core factorization were previously investigated by Liu [4, 5, 6], who focused on left-looking and multifrontal algorithms. Rothberg and Schreiber [10] implemented left-looking and multifrontal algorithms, as well as hybrids (combinations of the first two), considering that only the factor entries are stored externally but allowing multiple read and write operations. An earlier out-of-core multifrontal implementation is discussed in [2].

The work presented in this paper continues the study that we previously discussed in [3], where we focused on complexity analysis. Here we report new experimental results.

2 The Algorithms

An in-core factorization allocates storage for the whole of A and L. This allocation is performed at the beginning of the computation and it is kept until the computation ends. During the computation modifications in the storage allocation are allowed only for temporary data. For an in-core algorithm the system's memory manager can manage the whole storage allocation.

A general out-of-core factorization assumes that both A and L are stored externally and continuously adapts the core allocation in order to store fragments of A and L and temporary data. In our current research we focus on a particular case in which only the entries of L are stored externally (storage is still allocated for the whole of A and the indices of L) and the external traffic is minimized. This is a reasonable assumption because for a large number of applications the entries of L require significantly more storage than anything else (considering that the indices of L are stored in compressed form).

Some of the storage allocation can still be managed by the system's memory manager but an out-of-core algorithm may require a specialized core manager as well. Imagine the core as being split into two parts, one that is used to store all the symbolic data and the entries of A, and one that is used to store the entries of L as well as the temporary entries. We will refer to the latter as the numerical core (even if the former stores entries as well). The system's memory manager can then be used to allocate the numerical core as part of the whole core and the specialized manager is used to manage the numerical core.

Since the entries of L are the only data that are stored externally, the external traffic is minimized if each entry of L is written exactly once, discarded from the core when no longer needed and never read back. To perform an out-of-core factorization under such conditions, certain entries must be kept in core at certain phases of the factorization. For the left-looking and right-looking factorizations these are factor entries that were already accessed and need to be accessed in the future as well. For the multifrontal factorization

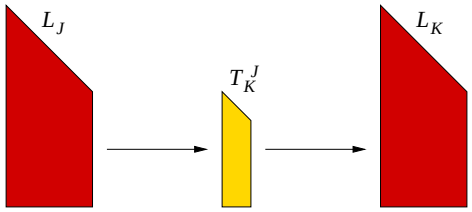

Figure 1. *Left-looking/right-looking factorization.*

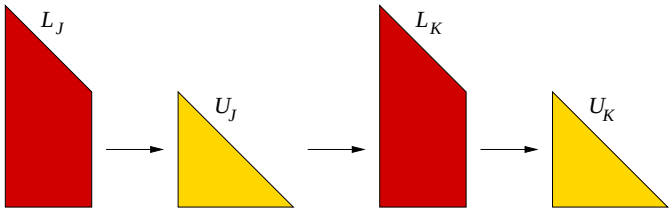

Figure 2. *Multifrontal factorization.*

these are the temporary entries that form the update stack. Keeping these entries in-core determines a minimum core size and a certain amount of internal data traffic (the latter is due to the core reorganizations).

We are interested in supernode-supernode factorization algorithms, which exploit the cache hierarchy efficiently [9]. We denote supernodes using capital letters such as J and K. The corresponding groups of columns of L and A are denoted as L_J and L_K, and A_J and A_K, respectively. In the left-looking and right-looking algorithms the temporary data that carries the updates between supernodes J and K are denoted as T_K^J. In the multifrontal algorithm the temporary data associated with supernode J (the update matrix associated with supernode J) are denoted as U_J. For convenience we also use F_J in order to refer to the frontal matrix associated with supernode J, but note that F_J is just an alias for the association of L_J and U_J. We express the algorithms in terms of basic operations: initialization (ZERO), scattered addition (ASSEMBLE), supernodal factorization (FACTOR), supernodal update (UPDATE), memory management (ALLOCATE, DISCARD, REORGANIZE), I/O (WRITE), and termination (ABORT).

In addition, we use the following sets: $\mathcal{F}(J)$ - the set of indices of the columns associated with supernode J; $\mathcal{U}(J)$ - the set of indices of those columns updated by the columns with indices in $\mathcal{F}(J)$; $\mathcal{C}(J)$ - the set of children of supernode J in the elimination tree [7] associated with L.

Figure 1 is a conceptual illustration of the left-looking and right-looking factorization algorithms. It describes the relationship between factor and temporary entries. The same happens in Fig. 2, which depicts the multifrontal factorization algorithm.

Algorithm 1 represents the minimum I/O out-of-core left-looking factorization. The corresponding right-looking factorization, not shown, can be obtained by reversing the

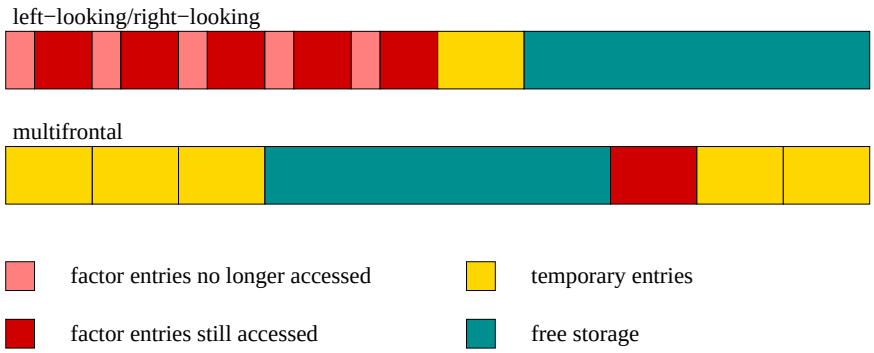

Figure 3. *The numerical core allocation.*

loop order. Both algorithms follow the idea used in [6], keeping mainly factor entries in-core. These are stacked at the beginning of the numerical core. Storage is allocated for factor entries the first time they are accessed and they are kept in-core as long as they are still going to be accessed. Since temporary entries are needed only while performing supernodal updates, they can be briefly stored on top of the stack of factor entries. This allocation policy is illustrated in Fig. 3.

The ALLOCATE and DISCARD operations are implemented by the specialized core manager, not by the system's memory manager. Note that only temporary entries are explicitly discarded. Factor entries are discarded during core reorganizations. A core reorganization compresses the stack of factor entries by discarding the entries that are no longer going to be accessed and by shifting the remaining entries toward the beginning of the numerical core (which determines the internal traffic).

Algorithm 2 represents the minimum I/O out-of-core multifrontal factorization. This keeps mainly temporary entries (the update stack) in-core. An interesting feature of the out-of-core multifrontal algorithm is that it does not require core reorganizations. This is possible by distinguishing the supernodes with an even depth in the elimination tree from the supernodes with an odd depth in the elimination tree, following the idea used in [10]. If we stack the update matrices associated with the former type at the beginning of the numerical core and the update matrices associated with the latter type at the end of the numerical core, the stack is always compressed (at both ends of the numerical core). Since factor entries are needed only during the processing of a supernode, they can be briefly stored next to corresponding update matrix. Figure 3 illustrates this allocation policy as well.

Again, the ALLOCATE and DISCARD operations are implemented by the specialized core manager, but since there are no core reorganizations, this time both factor and temporary entries are explicitly discarded.

```
1.    for K := 1 to N begin
2.        if no room for L_K
3.            REORGANIZE();
4.        if no room for L_K
5.            ABORT();
6.        ALLOCATE(L_K);
7.        ZERO(L_K);
8.        ASSEMBLE(A_K,L_K);
9.        for J < K and U(J) ∩ F(K) ≠ ∅ begin
10.           if no room for T_K^J
11.               REORGANIZE();
12.           if no room for T_K^J
13.               ABORT();
14.           ALLOCATE(T_K^J);
15.           ZERO(T_K^J);
16.           UPDATE(L_J,T_K^J);
17.           ASSEMBLE(T_K^J,L_K);
18.           DISCARD(T_K^J);
19.       end
20.       FACTOR(L_K);
21.       WRITE(L_K);
22.   end
```

Algorithm 1. *Out-of-core left-looking factorization with minimum I/O. Note that entries from each supernode L_J are discarded from the core only after all corresponding updates to higher numbered supernodes are performed, and hence do not need to be read after they are written.*

3 The Minimum Size of the Core and the Amount of Internal Traffic

Algorithms that compute the minimum size of the core and the amount of internal traffic can easily be implemented, basically by simulating the factorization.

In order to compute the minimum core size we need to keep track of the number of in-core factor/temporary entries. For left-looking and right-looking factorizations we also need to assume that the numerical core is reorganized after each FACTOR and UPDATE operation. This way entries are discarded as soon as they are no longer going to be accessed.

In order to compute the amount of internal traffic we need to keep track of the number of factor entries that are moved within the core (remember that no internal traffic is required for the multifrontal factorization).

The algorithms that compute the minimum size of the core and the amount of internal traffic are implemented by accessing only symbolic data, therefore they require significantly less storage than the actual factorizations. For example, for 2d and 3d problems ordered with nested dissection, their space complexity is $\Theta(n)$, where n is the order of the

```
1.    for K := 1 to N begin
2.        if no room for L_K
3.            ABORT();
4.        ALLOCATE(L_K);
5.        ZERO(L_K);
6.        ASSEMBLE(A_K,L_K);
7.        if no room for U_K
8.            ABORT();
9.        ALLOCATE(U_K);
10.       ZERO(U_K);
11.       for J in C(K) begin
12.           ASSEMBLE(U_J,F_K);
13.           DISCARD(U_J);
14.       end
15.       FACTOR(L_K);
16.       UPDATE(L_K,U_K);
17.       WRITE(L_K);
18.       DISCARD(L_K);
19.   end
```

Algorithm 2. *Out-of-core multifrontal factorization with minimum I/O. Here entries from supernode L_J are discarded as soon as they are written since L_J updates only its parent supernode, and this update is stored in U_J.*

problem.

The algorithms are also fast. The time complexity of the algorithms that compute the minimum core size is $\Theta(m)$ for left-looking and right-looking factorization, where m is the number of edges in the supernodal graph, and $\Theta(n)$ for multifrontal factorization. The time complexity of the algorithms that compute the amount of internal traffic is generally difficult to determine because of the irregular nature of the core reorganizations. However, we noticed that the number of core reorganizations is relatively small within a large range of the numerical core size. If we approximate the number of core reorganizations as constant then the time complexity is $\Theta(m)$ for the algorithms that compute the amount of internal traffic as well.

In this paper we consider 2d and 3d model problems (ordered with nested dissection). The grid sizes are of the form $k = 2^l - 1$, where l is a positive integer. The problem size n, the number of equations or unknowns, is then $n = k^2$ for 2d and $n = k^3$ for 3d problems.

Figure 4 depicts the ratio between the factor size and the minimum core size for 9-point stencil 2d models (top) and 27-point stencil 3d models (bottom). We choose this ratio instead of the minimum core size in order to show the difference between the storage required by an in-core factorization and the storage required by an out-of-core factorization. The x-axis, drawn to logarithmic scale, corresponds to the problem size as defined in the previous paragraph.

Note the asymptotic behavior suggested by Fig. 4, which can be confirmed by tedious

analysis. For general 2d problems ordered with nested dissection the numerical core grows as $\Theta(n)$ (for all three algorithms) while the factor grows as $\Theta(n \log(n))$. Thus the larger the problem, the larger the factor/core ratio. For the largest problem considered ($k = 1,023$, $n = 1,046,529$) for example, we can perform a right-looking factorization with a numerical core that is about 21 times smaller than the factor. On the other hand 3d problems do no share this nice property. In this case both the numerical core and the factor grow as $\Theta(n^{4/3})$ (again, for all three algorithms) and the factor/core ratio flattens as n increases. This time for the largest problem considered ($k = 63$, $n = 250,047$), we can perform a right-looking factorization with a numerical core that is about 6 times smaller than the factor.

Among the three algorithms right-looking requires the smallest numerical core, while left-looking requires the largest one, with the multifrontal algorithm in the middle.

Figure 5 illustrates the amount of internal traffic for the largest 2d (top) and 3d (bottom) model problems from Fig. 4. The core size ranges from slightly larger than the largest minimum core to slightly smaller than the factor.

At minimum core size there are many reorganizations and the amount of internal traffic is very large. However, a slight increase of the core size from the minimum determines a significant decrease of both the number of reorganizations and the amount of internal traffic. After that they become reasonably small and eventually they drop to zero if the core size is increased beyond the factor size (in that case we could actually perform an in-core factorization). The irregular nature of the core reorganizations makes it unlikely that a complexity analysis could be performed for the amount internal traffic.

With less internal traffic the right-looking algorithm outperforms the left-looking algorithm again. This time, however, the multifrontal algorithm outperforms the other two, since it requires no internal traffic at all.

4 Conclusion

A minimum I/O factorization requires a minimum core size and an amount of internal traffic determined by the core size available. These two values can be determined for a particular problem by fast simulation algorithms.

We determined the relationship between the three major column-oriented factorization algorithms for 2d and 3d model problems. For this set of problems the right-looking factorization requires the smallest minimum core and the left-looking factorization requires the largest one. The right-looking factorization also requires fewer core reorganizations and less internal traffic than the left-looking factorization. However, with no internal traffic and a minimum core size between the other two, the multifrontal factorization is probably the best candidate.

The nature of the core management is what makes the multifrontal algorithm more appealing. For left-looking and right-looking factorization the core manager has to keep track of the entries that are no longer going to be accessed in order to discard them during a core reorganization. No such feature is required for the multifrontal algorithm. The multifrontal core manager must only allocate and free storage. In order to avoid internal traffic, this allocation must be performed at both ends of the numerical core, but this is easy to implement. On the other hand, if the internal traffic does not really affect the

performance of the factorization, we can directly use the system's memory manager. This is not possible for the left-looking and right-looking factorizations. Therefore, the minimum I/O multifrontal out-of-core factorization is the easiest to implement and, for a significant number of applications, it requires a reasonably small minimum core.

We also note that 2d problems problems are better suited for minimum I/O out-of-core factorization than 3d problems due to the lower space complexity of the minimum core size relative to the factor size. We are currently testing a multifrontal implementation and preliminary results indicate that we are able to significantly extend the size of the 2d problems we solve while preserving high performance.

We are aware of problems for which the multifrontal algorithm does not perform well and we plan to include such problems in future tests, among various problems coming from real applications. As we mentioned, we have currently implemented the multifrontal algorithm but we would like to implement the other two as well in order to compare them in practice as well.

Ultimately, this study sets the path toward a general out-of-core factorization, which would require more I/O traffic. An important question in this case is which of the three algorithms considered here will reduce the I/O traffic. The design of hybrid algorithms that employ the algorithms considered here at different phases of the computation is one attractive potential approach [10].

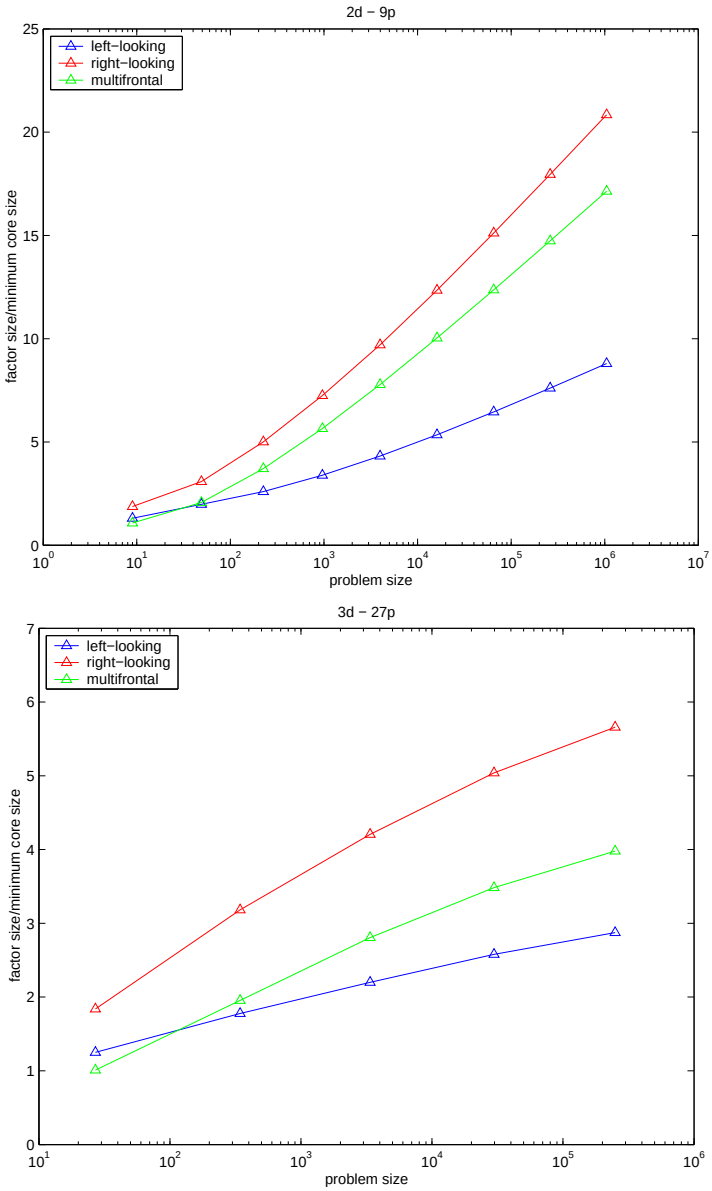

Figure 4. *The ratio between the factor size and the minimum core size as a function of the problem size for 2d (top) and 3d (bottom) model problems.*

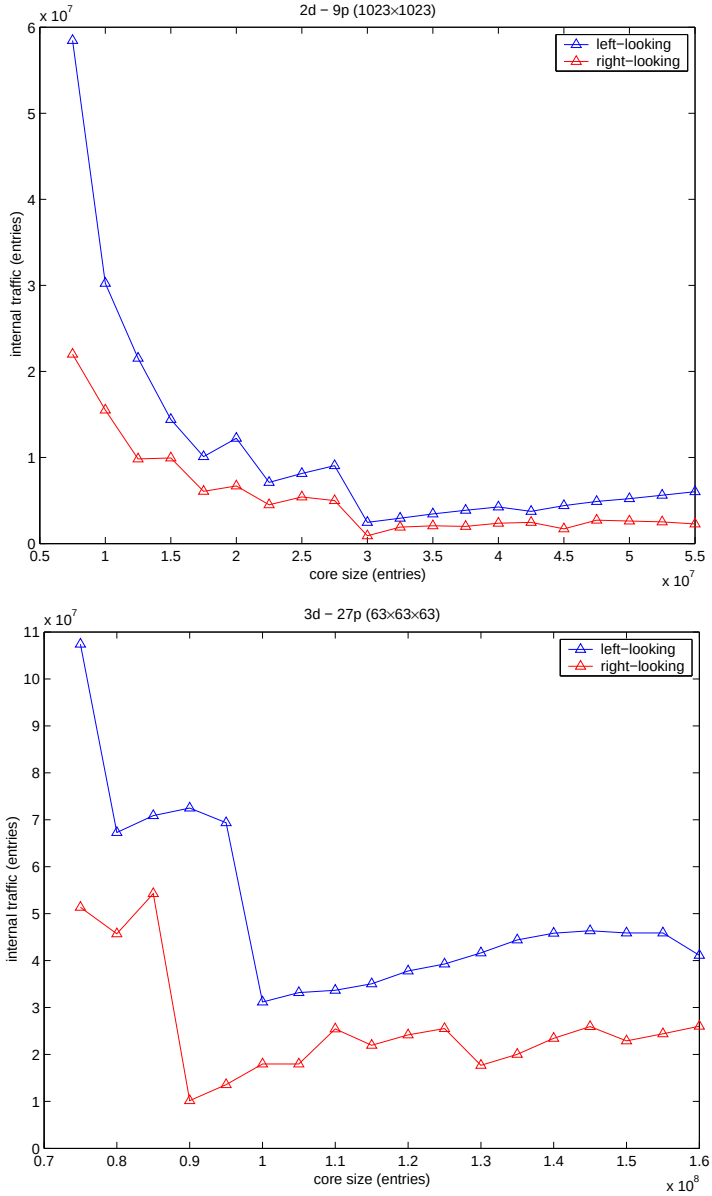

Figure 5. *The amount of internal traffic due to core reorganizations for the left-looking and right-looking algorithms, for a 2d problem (top) and a 3d problem (bottom). We organize multifrontal algorithms to require zero internal traffic.*

Bibliography

[1] J. M. ABELLO AND J. S. VITTER, eds., *External Memory Algorithms*, American Mathematical Society, 1999.

[2] C. ASHCRAFT, R. GRIMES, AND J. LEWIS, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, International Journal of Supercomputer Applications, 1 (1987), pp. 10–30.

[3] F. DOBRIAN, *External Memory Algorithms for Factoring Sparse Matrices*, PhD thesis, Old Dominion University, 2001.

[4] J. W. H. LIU, *An application of generalized tree pebbling to sparse matrix factorization*, tech. rep., York University, Apr. 1986.

[5] ——, *On the storage requirement in the out-of-core multifrontal method for sparse factorization*, ACM Transactions on Mathematical Software, 12 (1986), pp. 249–264.

[6] ——, *An adaptive general sparse out-of-core Cholesky factorization scheme*, SIAM Journal on Scientific and Statistical Computing, 8 (1987), pp. 585–599.

[7] ——, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[8] ——, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Review, 34 (1992), pp. 82–109.

[9] E. G. NG AND B. W. PEYTON, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM Journal on Scientific Computing, 14 (1993), pp. 1034–1056.

[10] E. ROTHBERG AND R. SCHREIBER, *Efficient methods for out-of-core sparse Cholesky factorization*, SIAM Journal on Scientific Computing, 21 (1999), pp. 129–144.

[11] S. TOLEDO, *A survey of out-of-core algorithms in numerical linear algebra*, in External Memory Algorithms, J. M. Abello and J. S. Vitter, eds., vol. 50 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999, pp. 161–179.