

Oblio: Design and Performance

Florin Dobrian and Alex Pothen*

Department of Computer Science and Center for Computational Sciences,
Old Dominion University,
Norfolk, VA 23529, USA
{dobrian, pothen}@cs.odu.edu

Abstract. We discuss Oblio, our library for solving sparse symmetric linear systems of equations by direct methods. The code was implemented with two goals in mind: efficiency and flexibility. These were achieved through careful design, combining good algorithmic techniques with modern software engineering. Here we describe the major design issues and we illustrate the performance of the library.

1 Introduction

We describe the design of Oblio, a library for solving sparse symmetric linear systems of equations by direct methods, and we illustrate its performance through results obtained on a set of test problems.

Oblio was designed as a framework for the quick prototyping and testing of algorithmic ideas [3] (see also [1, 5, 10, 11] for recent developments in sparse direct solvers). We needed a code that is easy to understand, maintain and modify, features that are not traditionally characteristic of scientific computing software. We also wanted to achieve good performance. As a consequence, we set two goals in our design: efficiency and flexibility.

Oblio offers several options that allow the user to run various experiments. This way one can choose the combination of algorithms, data structures and strategies that yield the best performance for a given problem on a given computational platform. Such experimentation is facilitated by decoupled software components with clean interfaces. The flexibility determined by the software design is Oblio's major strength compared to other similar packages.

Out of the three major computational phases of a sparse direct solver, *pre-processing*, *factorization* and *triangular solve*, only the last two are currently implemented in Oblio. For preprocessing we rely on external packages.

2 Design

Factorization Types. The first major choice in Oblio is the factorization type. Generally, the direct solution of a linear system $Ax = b$ requires factoring a

* This research was partially supported by U.S. NSF grant ACI-0203722, U.S. DOE grant DE-FC02-01ER25476 and LLNL subcontract B542604.

permutation of the coefficient matrix A into a product of lower and upper triangular matrices (factors) L and U (L unit triangular). If A is symmetric then U can be further factored as DL^T , where D is generally block diagonal, and if A is positive definite as well then we can write $A = LU = LDL^T = \tilde{L}\tilde{L}^T$ (D is purely diagonal in this case), where \tilde{L} is lower triangular. The latter decomposition of A is the Cholesky factorization and \tilde{L} is the Cholesky factor.

In Oblio we offer three types of factorizations. The first one is the Cholesky factorization $\tilde{L}\tilde{L}^T$, which does not require pivoting and therefore has a static nature. The other two are more general LDL^T factorizations. One of them is static, the other one is dynamic.

The static LDL^T factorization does not usually allow row/column swaps (*dynamic pivoting*). Instead, it relies on small perturbations of the numerical entries (*static pivoting*) [7]. Although this changes the problem, it is possible to recover solution accuracy through *iterative refinement*. The static LDL^T factorization can be, however, enhanced, by allowing dynamic pivoting as long as this does not require modifications of the data structures. In Oblio we provide the framework for both approaches.

Numerical preprocessing is generally required before a static LDL^T factorization in order to reduce the number of perturbations as well as the number of steps of iterative refinement [4]. Since the numerical preprocessing of sparse symmetric problems is currently an area of active research, this type of preprocessing is not yet present in Oblio.

The dynamic LDL^T factorization performs dynamic pivoting using 1×1 and 2×2 pivots and can modify the data structures. Two pivot search strategies are currently available, one biased toward 1×1 pivots and one biased toward 2×2 pivots [2]. Note that the dynamic LDL^T factorization can also benefit from numerical preprocessing: a better pivot order at the beginning of the factorization can reduce the number of row/column swaps.

For positive definite matrices the best choice is the $\tilde{L}\tilde{L}^T$ factorization. For matrices that are numerically more difficult one must choose between the two LDL^T factorizations. The dynamic LDL^T factorization is more expensive but also more accurate. It should be used for those matrices that are numerically the most difficult. For less difficult matrices the static LDL^T factorization might be a better choice.

Factorization Algorithms. The second major choice in Oblio is the factorization algorithm. We offer three major column based supernodal algorithms: *left-looking*, *right-looking* and *multifrontal*.

These three factorization algorithms perform the same basic operations. The differences come from the way these operations are scheduled. In addition to coefficient matrix and factor data these algorithms also manage temporary data. For left-looking and right-looking factorization this is required only for better data reuse, but for multifrontal factorization temporary data are also imposed by the way the operations are scheduled.

The multifrontal algorithm generally requires more storage than the other two algorithms and the difference comes from the amount of temporary data. If

this is relatively small compared to the amount of factor data then the multi-frontal algorithm is a good choice. The right-looking and multifrontal algorithms perform the basic operations at a coarser granularity than the left-looking algorithm, which increases the potential for data reuse. The right-looking algorithm is not expected to perform well in an out-of-core context [9].

Factor Data Structures. The third major choice in Oblio is the factor data structure (for $\tilde{L}\tilde{L}^T$ factorization this stores \tilde{L} ; for LDL^T factorization this stores L and D together). A static factorization can be implemented with a static data structure while a dynamic factorization requires a dynamic data structure. We offer both alternatives, although a static factorization can be implemented with a dynamic data structure as well.

The difference between the two data structures comes from the way storage is allocated for supernodes. In a static context a supernode does not change its dimensions and therefore a large chunk of storage can be allocated for all supernodes. In a dynamic context a supernode can change its dimensions (due to delayed columns during pivoting) and in this case storage needs to be allocated separately for each supernode. In order to access the data we use the same pointer variables in both cases. This makes the data access uniform between implementations.

The two data structures manage in-core data and therefore we refer to them as *in-core static* and *in-core dynamic*. In addition, we offer a third, *out-of-core*, data structure that extends the in-core dynamic data structure. The out-of-core data structure can store the factor entries both internally and externally. Of course, most of the factor entries are stored externally. Data transfers between the two storage layers are performed through I/O operations at the granularity of a supernode. A supernode is stored in-core only when the factorization needs to access it.

Other Features. A few other features are of interest in Oblio. By using a dynamic data structure for the elimination forest [8] it is easy to reorder for storage optimization and to amalgamate supernodes for faster computations. Iterative refinement is available in order to recover solution accuracy after perturbing diagonal entries during a static LDL^T factorization or after a dynamic LDL^T factorization that uses a relaxed pivoting threshold. Oblio can also factor singular matrices and solve systems with multiple right hand sides.

Implementation. Oblio is written in C++ and uses techniques such as dynamic memory allocation, encapsulation, polymorphism, and templates. Dynamic memory allocation is especially useful for temporary data. In order to minimize the coding effort we combine it with encapsulation most of the time, performing memory management within constructors and destructors. We use polymorphism in order to simplify the interaction between different factorization algorithms and different factor data structures, and we rely on templates in order to instantiate real and complex valued versions of the code.

Oblio is organized as a collection of *data* and *algorithm* classes. Data classes describe passive objects such as coefficient matrices, vectors, permutations and factors. Algorithm classes describe active objects such as factorizations and tri-

angular solves. The major classes and the interactions between them are depicted in Fig. 1 (real valued only).

We discuss now the interaction between the three factorization algorithms and the three factor data structures in more detail. Generally we would require three different implementations for each algorithm and thus a total of nine implementations. That would determine a significant programming effort, especially for maintaining the code. We rely on polymorphism instead and use only three implementations, one for each algorithm. Remember that the difference between the two in-core data structures comes from the supernode storage allocation. But as long as we can use a common set of pointers in order to access data, a factorization algorithm does not need to be aware of the actual storage allocation. Therefore the factorization algorithms interact with an abstract factor class that allows them to access factor data without being aware of the particular factor implementation. The whole interaction takes place through abstract methods.

For out-of-core factorization we make a trade-off. In addition to the basic operations that are required by an in-core factorization, an out-of-core factorization requires I/O operations. These can be made abstract as well but that would not necessarily be an elegant software design solution. Another solution, requiring additional code but more elegant, is to use run time type identification. This is the solution that we adopted in Oblio. Oblio identifies the data structure at run time and performs I/O only in the out-of-core context. The identification of the actual data structure is done at the highest level and thus virtually comes at no cost.

In Oblio, most of the basic arithmetic operations are efficiently implemented through level 3 BLAS/LAPACK calls.

3 Performance

We present a reduced set of results here. More details will be available in a future paper. We also refer the reader to [12] for a recent comparative study of sparse symmetric direct solvers (including Oblio). In all the experiments below we use the node nested dissection algorithm from METIS [6] as a fill reducing ordering.

As reported in [12] Oblio performs as well as other similar solvers for positive definite problems and correctly solves indefinite problems. For the latter Oblio is not as fast as other indefinite solvers since we have not yet optimized our indefinite kernels. The best factorization rate observed so far is 42% of the peak rate on an IBM Power 4 platform (1.3 GHz, 5200 Mflop/s peak rate).

In-Core Results. Figures 2 through 6 illustrate in-core experiments performed with Oblio. All problems are indefinite and represent a subset of the collection used in [12]. We used dynamic LDL^T factorization and, for comparison, we used $\tilde{L}\tilde{L}$ factorization as well, after replacing the original numerical entries and transforming indefinite matrices into positive definite matrices. We used the in-core dynamic factor data structure for the former and the in-core static factor data structure for the latter, and we employed all three factorization algorithms. The results are obtained on an IBM Power 3 platform (375 MHz, 16

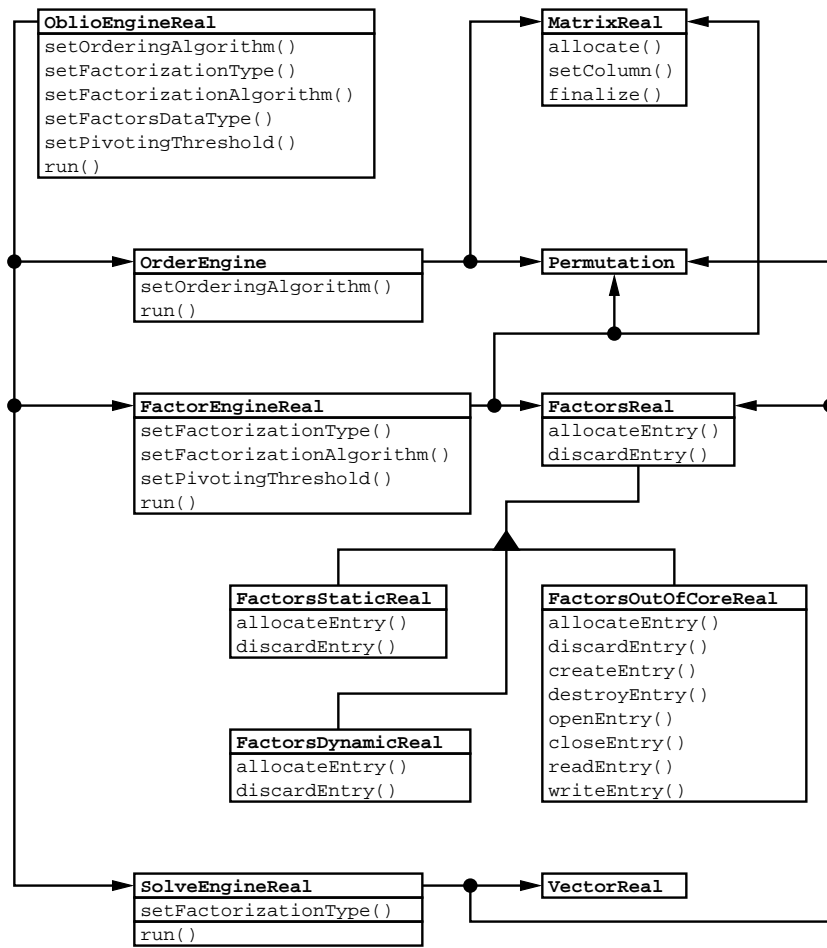


Fig. 1. The major classes from Oblio and the interactions between them (only the real valued numerical classes are shown but their complex valued counterparts are present in Oblio as well). The factorization algorithms interact with the abstract class `FactorsReal`. The actual class that describes the factors can be `FactorsStaticReal`, `FactorsDynamicReal` or `FactorsOutOfCoreReal` (all three derived from `FactorsReal`), the selection being made at run time.

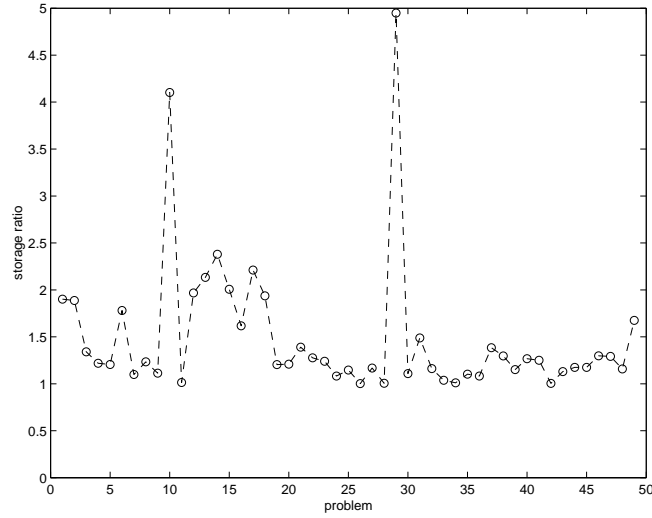


Fig. 2. The ratio between the storage required by the multifrontal algorithm and the storage required by the left-looking and right-looking algorithms, $\tilde{L}\tilde{L}^T$ factorization.

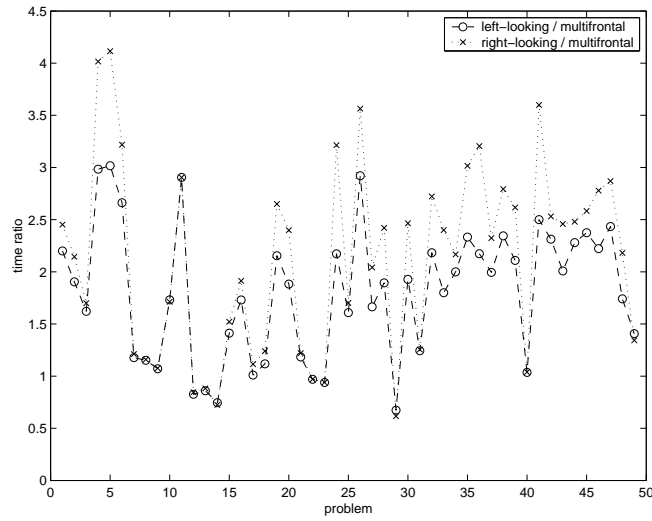


Fig. 3. The ratios between the execution times of the left-looking and multifrontal algorithms, and between the execution times of the right-looking and multifrontal algorithms, $\tilde{L}\tilde{L}^T$ factorization (IBM Power 3 platform: 375 MHz, 16 GB).

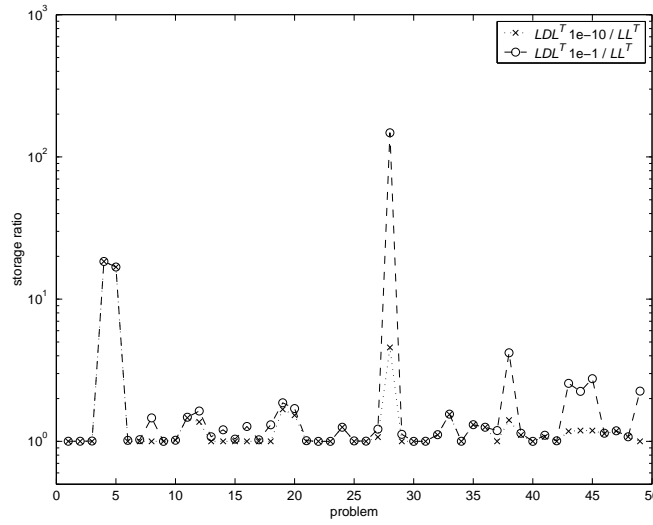


Fig. 4. The ratio between the storage required by the dynamic LDL^T factorization and the storage required by the $\tilde{L}\tilde{L}^T$ factorization, using the multifrontal algorithm, for pivoting thresholds of $1e-1$ and $1e-10$, respectively.

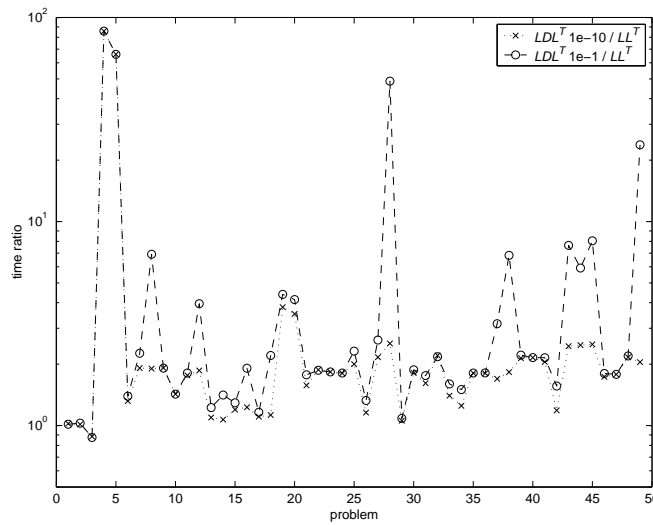


Fig. 5. The ratio between the execution time of the dynamic LDL^T factorization and the execution time of the $\tilde{L}\tilde{L}^T$ factorization, using the multifrontal algorithm, for pivoting thresholds of $1e-1$ and $1e-10$, respectively (IBM Power 3 platform: 375 MHz, 16 GB).

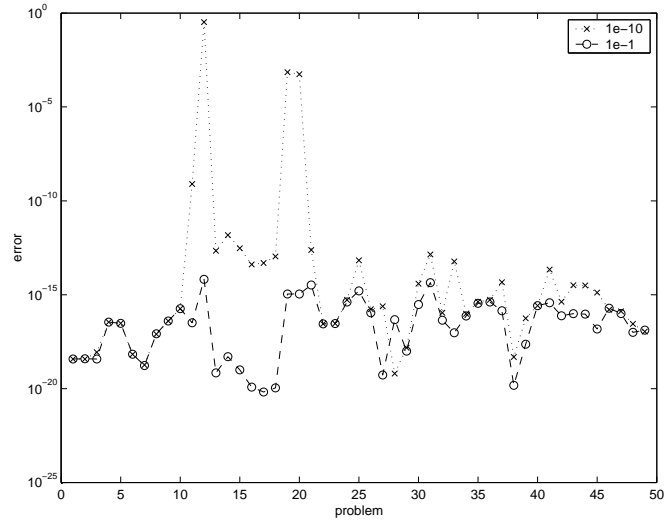


Fig. 6. The relative residual for the dynamic LDL^T factorization, using the multifrontal algorithm, for pivoting thresholds of $1e-1$ and $1e-10$, respectively.

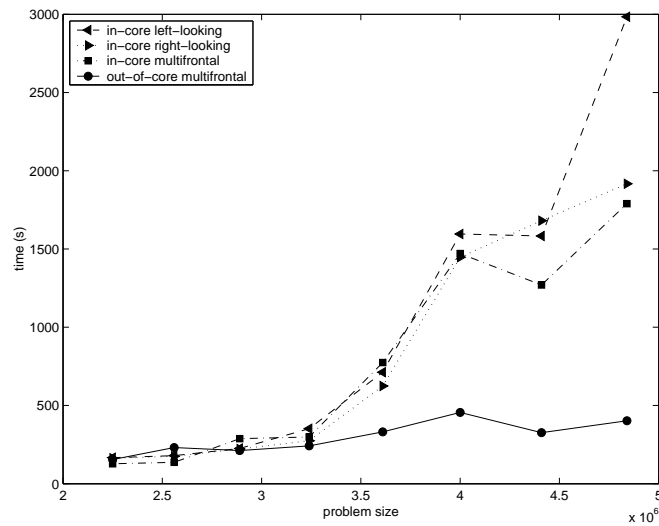


Fig. 7. The execution time of the $\tilde{L}\tilde{L}^T$ factorization, in-core (left-looking, right-looking and multifrontal) and out-of-core (multifrontal) (Sun UltraSparc III platform: 900 MHz, 2GB). There is only a slight increase of the out-of-core execution time because a slight increase of the problem size determines a slight increase of the amount of explicit I/O. The non-monotonicity of the plots is caused by the lack of smoothness in the ordering algorithm.

GB), which we chose because we were particularly interested in a large amount of internal memory.

Figure 2 shows the difference in storage requirement between the three algorithms, in the positive definite case. The plot represents the ratio between the storage required by the multifrontal algorithm and the storage required by the left-looking and right-looking algorithms. This indicates how much additional storage is required by the multifrontal algorithm. For a significant number of problems the amount of additional storage is not large and the multifrontal algorithm is expected to perform well.

Execution time results for positive definite problems are shown in Fig. 3. The plots represent the ratios between the times required to factor using the left-looking and multifrontal algorithms, and between the times required to factor using the right-looking and multifrontal algorithms, respectively. Note that the multifrontal algorithm outperforms the other two for many problems.

Figures 4 and 5 illustrate results for the indefinite case, using only the multifrontal algorithm and two different pivoting thresholds, 1e-1 and 1e-10. The plots show the increase in storage and execution time (same IBM Power 3 platform). The increase is smaller when the pivoting threshold is smaller but a larger pivoting threshold may be required for accurate results. Similar results can be obtained with the indefinite left-looking and right-looking algorithms.

For indefinite problems we also report accuracy results. These are provided in Fig. 6. The plots represent the relative residual for the two sets of experiments. For most of the problems from this set the computation is accurate enough with the relaxed pivoting threshold (1e-10) but some problems require a larger pivoting threshold, at the cost of a more expensive computation.

Out-of-Core Results. For the out-of-core experiments we chose a platform with a smaller amount of internal memory: an UltraSparc III Sun Fire 280R (900 MHz, 2GB). Since this platform has a smaller core we can easily determine differences in performance between the in-core and the out-of-core computations.

Unfortunately, matrix collections usually provide problems of a given size, even if some of these problems may be large. In order to run out-of-core experiments one should be able to tune the problem size given a particular core size. Here, in order to match the 2 GB core size we tune the order of the problem from roughly 2,000,000 to roughly 5,000,000 (this way we cross the 2GB threshold). We use model 2d finite difference discretization grids with 5-point stencils, the grid size ranging from 1,500 to 2,200.

Figure 7 plots the execution time for the three in-core $\tilde{L}\tilde{L}^T$ factorizations (static data structure) as well as the execution time for the out-of-core multifrontal $\tilde{L}\tilde{L}^T$ factorization. As expected, the in-core factorizations start to perform poorly close to the 2GB threshold. At that point the operating system begins to rely on virtual memory in order to provide the required storage. This translates into *implicit I/O* (swapping), which slows down the computation. On the other hand, the out-of-core factorization performs *explicit I/O* and therefore the computation continues to be performed efficiently.

4 Conclusion

Oblio is an active project and the package continues to be developed. Our current effort is targeted toward software design improvements and code optimizations. We also plan several extensions of the package, mostly in order to provide more options for numerically difficult problems. A future, longer paper will address the design of Oblio in more detail and will provide more results.

References

1. P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
2. C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM Journal on Matrix Analysis and Applications*, 20(2):513–561, 1998.
3. F. Dobrian, G. K. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. In H. P. Langtangen, A. M. Bruaset, and E. Quak, editors, *Advances in Software Tools in Scientific Computing*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 89–131. Springer-Verlag, 2000.
4. I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.
5. A. Gupta, M. Joshi, and V. Kumar. WSMP: A high-performance shared- and distributed-memory parallel sparse linear equation solver. Technical Report RC 22038, IBM T.J. Watson Research Center, 2001.
6. G. Karypis and V. Kumar. METIS: Family of multilevel partitioning algorithms. <http://www-users.cs.umn.edu/~karypis/metis>.
7. X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, 2003.
8. A. Pothen and S. Toledo. Elimination structures in scientific computing. In D. Mehta and S. Sahni, editors, *Handbook on Data Structures and Applications*, pages 59.1–59.29. CRC Press, 2004.
9. E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.
10. V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30(1):19–46, 2004.
11. O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.
12. J. A. Scott, Y. Hu, and N. I. M. Gould. An evaluation of sparse direct symmetric solvers: an introduction and preliminary findings. Numerical Analysis Internal Report 2004-1, Rutherford Appleton Laboratory, 2004.