Oblio: An Algorithmic Laboratory for Sparse Direct Solvers

Florin Dobrian[1], and Alex Pothen[1,2]

[1] Department of Computer Science, Old Dominion University
[2] ICASE, NASA Langley Research Center

Abstract. We present **Oblio**, a sparse direct solver library running in both serial an parallel environments. The code is written in C++ using object-oriented techniques, with the exception of few computationally intensive kernels that are written in Fortran 77. In this paper we explain what motivated the project, discuss design issues and report recent results.

1 Introduction

Solving sparse linear systems of equations represents the key computation in several critical industry and national-security applications ([1]). Two types of solvers exist: *direct* and *iterative*. We currently investigate the former, which are more robust, but which require more computational resources and are more difficult to program. Sparse direct solvers also represent an active area of research in the scientific computing community, new algorithms being developed continually.

Modern scientific computing needs to address two major issues ([2]): *complexity* and *efficiency*. Complexity is given and we must manage it. The complexity of the scientific computing software is determined by the sophistication of the *algorithms* involved but also by the complexity of the *applications* and *architectures*. Sparse direct solvers are representative from the first perspective because they use highly sophisticated algorithms, including irregular and dynamic computations. In addition, in terms of applications, the systems to be solved may be unsymmetric, structurally symmetric or fully symmetric, real or complex valued, and pivoting may or may not be required for numerical stability, while, in terms of architectures, memory hierarchies determine cache-aware and out-of-core computations and parallelism leads to the consideration of proper mapping and communication models.

As opposed to complexity, efficiency is a target which we attempt to achieve. An important observation is that efficiency has a *computer* component as well as a *human* component. From the computer perspective, the objective is to minimize execution time. From the human perspective, the objective is to minimize the time spent implementing and maintaining code. Until recently, the focus was mostly on the computer component but, as scientific computing software becomes increasingly complex, the human component started to receive more attention.

Our objective is to build modern software tools for sparse direct solvers, in such a way that human time is reduced while computer time remains constant. The immediate implications are that experiments would be performed faster if existing algorithms can be easily combined in different ways and if new algorithms can be quickly prototyped, and it would take less time to adapt software to various application and architectural requirements.

We have implemented **Oblio**, a sparse direct solver library that runs in both serial and parallel environments. The software is designed using object-oriented techniques and it is written in C++, except for few low level computationally intensive kernels which are written in Fortran 77. Currently, **Oblio** handles fully symmetric systems, both real and complex valued, and both positive definite and indefinite. In our opinion, the code is fairly easy to understand, modify and extend, and thus suited for algorithmic experimentation.

Our work is part of the current efforts to introduce advanced software design to the scientific computing community. Currently, iterative solvers are the main targets for the new software techniques, PETSc (written in C) ([5]) and Diffpack (written in C++) ([6]) being well known examples. Less has been done to improve the design of direct solvers. Spindle ([8, 11]) handles ordering algorithms (**Oblio** has only factorization and solve algorithms). We are aware of only one other sparse direct solver package that uses object-oriented design: SPOOLES ([3]). The major difference comes from the fact that SPOOLES is written in C. While additional programming effort and discipline are required in C, there is no doubt that C++ has portability problems, as a standard was only recently established and it is not rigorously followed. Such problems will likely disappear in the future.

We have presented earlier, serial only, work on **Oblio** in [7, 8]. In this paper we discuss the continuation of that work, highlighting support for parallel computations.

2 Problem Overview

Direct methods use a simple idea to compute the solution of a general linear system of equations $\boldsymbol{Ax} = \boldsymbol{b}$: decompose it into a sequence of systems which are easier to solve. This is equivalent to decomposing the coefficient matrix into a product of simpler matrices. The procedure is called factorization and the matrices computed by it are called factors. Factorization algorithms are commonly variations of the Gaussian elimination, the factors having triangular and diagonal shapes. The task of solving triangular and diagonal systems is obvious.

A sparse symmetric matrix is usually decomposed as $\boldsymbol{PAP^T} = \boldsymbol{LDL^T}$ where $\boldsymbol{P}$ is a matrix that permutes $\boldsymbol{A}$ to preserve sparsity after the factorization, and $\boldsymbol{L}$ and $\boldsymbol{D}$ are factors with triangular and diagonal shapes respectively. Details about this procedure are available in [10]. In the remainder of this section we consider that the coefficient matrix has been permuted and we focus on factorization, the most interesting computation from the parallelization perspective.

Sparse matrix computations can be suggestively illustrated in terms of graph algorithms. Figure 1 shows a sparse symmetric matrix and its associated graph $G(\boldsymbol{A})$. Similarly, Fig. 2 depicts the corresponding *filled matrix* ($\boldsymbol{F} = \boldsymbol{L}+\boldsymbol{D}+\boldsymbol{L^T}$) and *filled graph* ($G^+(\boldsymbol{A})$), which result after the factorization.

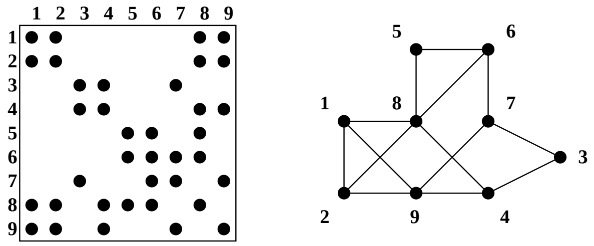

Fig. 1. A sparse symmetric matrix and its associated graph.

The factorization is organized along the *elimination forest*, the transitive reduction of the fill graph ([12]). The forest is useful in defining criteria for increasing performance and exploiting parallelism. First, sparsity leads to irregular computations, which are inefficient. However, columns along forest branches have identical or similar sparsity patterns and they can be clustered into *fronts*. Within a front computations become regular and thus performance increases. Second, parallelism can be exploited in two ways. With *branch level parallelism* independent branches can be processed concurrently. With *front level parallelism* update operations between columns are carried out in parallel. It is important to exploit both levels of parallelism because there are fewer independent branches and larger fronts at the higher layers of the elimination forest.

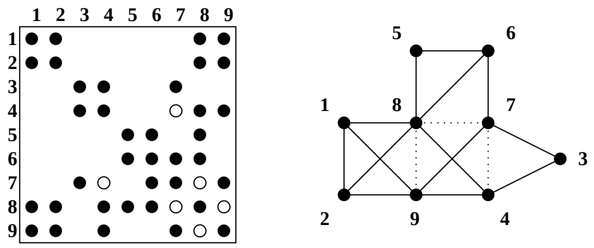

Fig. 2. The corresponding fill matrix and fill graph.

Out of several possible factorization strategies we use the *multifrontal* one, first mentioned in [9]. A good tutorial on multifrontal factorization in given in [13] and additional details about pivoting can be found in [4]. Basically, the computation is carried out by traversing the elimination forest in postorder and by means of *frontal* and *update* matrices (both dense matrices). Figure 3 shows the elimination forest that corresponds to the coefficient matrix from Figure 1 and the frontal and update matrices associated with the top nodes. Note how several nodes are clustered.

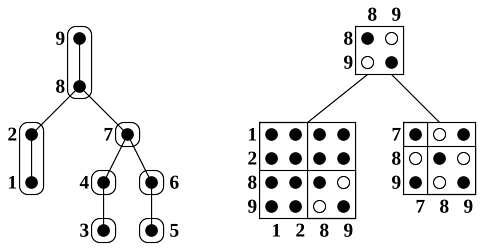

Fig. 3. The elimination forest, and the frontal and update matrices.

The common strategy used by a parallel solver during the factorization is based on a recursive mapping of the computation to processors. The processors are first split among the forest trees. This exploits branch level parallelism. A proportional mapping ([14]), which takes the work associated to each tree into account, is desirable. The front associated with the root of each tree is processed by the processors that tree is mapped on. This exploits front level parallelism. The procedure is then repeated recursively starting with the subtrees rooted at the children of each tree. Whenever only one processor is left, the whole subtree is mapped on it.

3 Software Design

In a parallel environment **Oblio** distributes the cost dominant data structures and replicates few others. The factorization arithmetic is fully distributed. Basically, each processor replicates the elimination forest and traverses it in postorder, exactly like in a serial environment ([8]). However, additional mapping information helps processors skip nodes whose processing they are not responsible for (branch level parallelism) or partially process nodes (front level parallelism). The coefficient matrix, the factors and the frontal and update matrices are distributed.

In this section we discuss the major components of **Oblio** and, in the end, we compare the design of **Oblio** with the design of SPOOLES.

3.1 Array Handling

Most **Oblio** objects store data in arrays. We use a customized `Array` data structure that couples a C++ array with its length. This way fewer lines of code need to be typed and inconsistency is avoided. Since items of different data types are stored, `Array` is a template class. Other classes make use of `Array` objects by storing pointers to them.

3.2 Coefficient Matrices and Factors

The objects visible at the highest level are the coefficient matrix and the factors. In **Oblio**, these are described by the `SparseMatrix` and `SparseFactors` data structures.

`SparseMatrix` is shown in Fig. 4. Because we solve both real and complex valued systems, `SparseMatrix` is a template class. The same is true for any other class that stores numerical data.

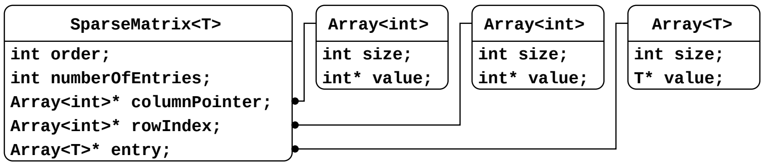

Fig. 4. The template class that describes coefficient matrices.

The dimensions of a `SparseMatrix` object are the order and the number of nonzero entries. In addition, three `Array` objects store column pointers, row indices and entries (compressed column format). While **Oblio** currently handles only symmetric systems, the whole coefficient matrix is stored. This speeds up certain computations, so we trade storage for execution time.

`SparseFactors` is very similar to `SparseMatrix`. In fact we could have used the same class for both the coefficient matrix and the combined factors (the filled matrix). Yet, there are reasons to use a separate class: a slightly different storage scheme increases performance and symmetry can be effectively exploited this time by saving half of the storage. Note that combining the factors in a single object determines some coupling, but the factors are always manipulated together.

In a parallel environment `SparseMatrix` and `SparseFactors` objects are actually fragments of the coefficient matrix and factors respectively. Additional maps translate between global and local data.

3.3 The Elimination Forest

An elimination forest object is described by the `EliminationForest` class, derived from the more general `Forest` class, as shown in Fig. 5. We use this approach to allow the specialization of other forest-like data structures.

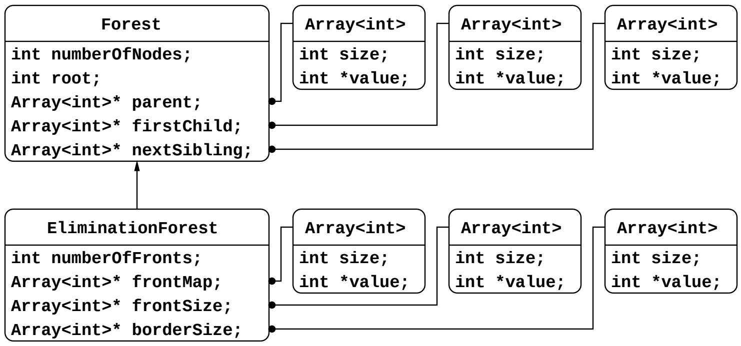

Fig. 5. The classes that describe forests and elimination forests.

A `Forest` object has a single dimension: the number of nodes. Three `Array` objects store the parent, first child and next sibling of each node. A second dimension is added by an `EliminationForest` object: the number of fronts. Also, three additional `Array` objects store a front map, and, for each front, its size (the number of columns in the front) and the size of its border (the number of columns updated by the front).

To provide a convenient mechanism to traverse `Forest` objects, iterator classes are associated with the `Forest` class. The factorization, for example, uses these iterators.

When **Oblio** runs in a parallel environment, all processors replicate the same `EliminationForest` object.

3.4 Factorization Specifics

The postorder traversal of the elimination forest represents the outer loop of the factorization and the place to exploit branch level parallelism by skipping unowned nodes. The core of the factorization is the processing performed at each node. This is the place where one must focus on performance and exploit front level parallelism. Frontal and update matrices are the key objects during the processing of a front.

Both frontal and update matrices are dense. The common description is captured by `DenseMatrix` and then `FrontalMatrix` and `UpdateMatrix` are derived from it. Since this is the place where most of the differences between serial and parallel factorization come from, we discuss both the initial approach and the current solution.

Figure 6 shows the solution used in the first version of **Oblio**. A `DenseMatrix` object has one dimension, the order. The entries are stored by an `Array` object. Two other `Array` objects store column pointers and a map that translates from local to global indices. Note that column pointers do not need to be stored explicitly here but doing so speeds up the factorization.

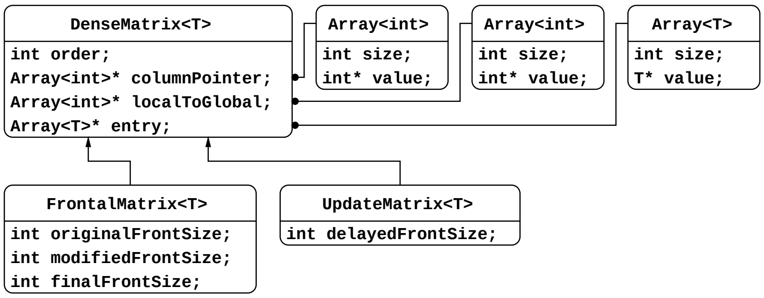

Fig. 6. Initial approach for frontal and update matrices.

`FrontalMatrix` and `UpdateMatrix` specialize `DenseMatrix` by adding dimensions and functionality. `FrontalMatrix` objects add three dimensions. The original front size, computed during the symbolic factorization, represents the actual size of the front if no pivoting is performed. The original front may be modified by pivoting. It increases if columns migrate between fronts. The modified front size includes these migrations and the final front size indicates the actual number of columns that were processed at a front. As for functionality, `FrontalMatrix` objects load and save entries and perform partial factorization. `UpdateMatrix` objects are much simpler, adding only one dimension, the size of the delayed front, and no functionality.

While this solution may be satisfactory in a serial solver, it is not useful in a parallel one. In addition there are some drawbacks if used in a serial code too. In a parallel code the goal is to distribute both work and storage. With the solution explained above we would be able to distribute only work. For a front mapped on more than one processor, each participating processor would allocate space for the full frontal and update matrices. Using an ownership map, a processor can skip unowned columns, but storage is wasted anyway.

The right thing to do in the parallel case is to split dense matrices into panels, each panel storing a certain number of columns. This way a processor allocates storage only for the columns it owns. Note that there are advantages in using this solution in the serial case too. As fronts become bigger at higher levels in the elimination forest, cache use becomes important and panels can make a big difference. Also, if there is a clear boundary between frontal panels and update panels, there is no need to copy entries from a frontal matrix to an update matrix. Only pointers to panels need to be be copied.

Panels have one disadvantage though: they restrict the pivot search space for indefinite systems. But there is no choice with parallel factorization. The only requirement is to determine the right panel size, which must be small enough to obtain good load balance and large enough to increase pivot search space as well as to reduce communication overhead.

Our current solution is shown in Fig. 7. Note that this is more general than the first one because we can quickly switch to our initial approach by choosing panels as wide as the order of the dense matrix they correspond to.

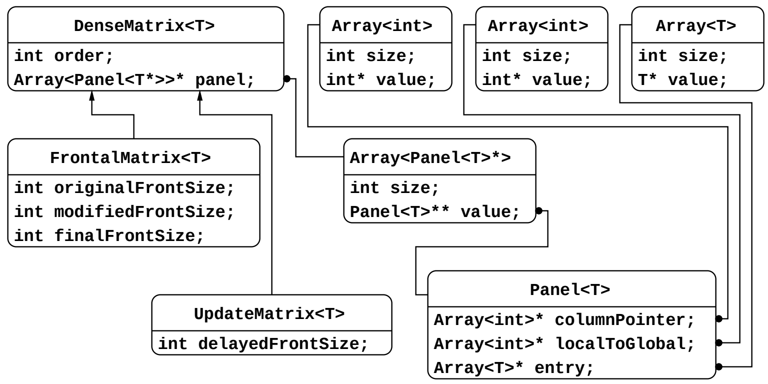

Fig. 7. Current solution for frontal and update matrices.

The factorization algorithm is basically the same as in the earlier version of **Oblio**. Details are available in [8]. The class that describes the multifrontal factorization is `MultifrontalFactor` and the computation is triggered by the `run` method associated with this class, which traverses the elimination forest in postorder and executes the operations required at each node. In a parallel environment each processor calls `MultifrontalFactor:run` but executes only the operations it is responsible for.

3.5 Comparison with SPOOLES

SPOOLES builds many of its classes based on data structures for arrays similar to those used in **Oblio**. However, because it is written in C, the same code needs to be replicated for each particular data type used for array items. This approach increases programming effort and it is subject to errors.

The data structures for the coefficient matrices, the permutations, the factors and the elimination forest are also similar. The decision of designing a general forest data strucure first helps because SPOOLES also specializes a domain/separator forest (in addition to the elimination forest) for ordering purposes.

Significant differences between **Oblio** and SPOOLES appear during the factorization. First, SPOOLES performs a left-looking factorization instead of a multifrontal one. Thus, only frontal matrix data structures are required. Second, fronts are not decomposed into panels. Instead, larger fronts are split into smaller ones. Although conceptually the same, few details make these two approaches slightly different. It is not clear yet which one is superior. Third, the factorization algorithm is associated with the data structure that describes the factor. In C++ this would be equivalent to making the factorization algorithm a method within a class. As explained in [8], having separate classes for certain algorithms is a more natural and flexible approach.

4 Results

In Table 1 we report recent results obtained on an SGI Origin 2000 multiprocessor (64 processors, 250 MHz, 16 GB RAM) running Irix 6.5. The code was compiled with g++ 2.8.1 (-O) and f77 7.2.1.1m (-O3) and the coefficient matrix was reordered using nested dissection. The test set contains model problems (grid2d9p255, grid2d9p511, grid3d7p31, grid3dt31), strucural analysis problems (glass2, bcsstk31, bcsstk32) and computational fluid dynamics problems (onera_dual, tandem_dual).

Table 1. Results on an SGI Origin 2000 multiprocessor.

problem	order	time (s)	speedup			
		1 P	2 P	4 P	8 P	16 P
grid2d9p255	65,025	3.97	1.81	3.01	3.85	4.67
grid2d9p511	261,121	27.34	1.89	3.28	4.86	5.94
glass2	17,037	30.45	1.37	3.03	5.82	7.73
grid3d7p31	29,791	53.96	1.69	3.09	5.38	8.15
grid3dt31	29,791	61.17	1.41	2.62	5.50	9.50
bcsstk31	35,588	14.41	1.34	2.11	3.69	5.72
bcsstk32	44,609	16.09	1.32	2.25	4.00	5.45
onera_dual	85,567	31.47	1.44	2.89	4.13	6.64
tandem_dual	94,069	29.66	1.95	2.67	4.11	5.18

The rather modest speedup is caused by two factors. First, we concentrated our efforts on design issues. Better results are expected as we will spend more time with performance tuning. Second, we currently use a one-dimensional mapping (along columns) of the computation and we do not overlap communication with computation. Theoretically, better results can be achieved with a two-dimensional mapping (along both columns and rows) and with overlaped communication ([15]). However, such conclusions are drawn using idealized models and may not hold in practice. A two-dimensional mapping for example may increase the cost of pivoting.

We plan to include both mapping and both communication models in the future versions of our code, to allow different combinations of experiments. It is important to remember that, by reducing human time, object-oriented design enables such experimentation.

References

[1] Anonymous. U.S. has sparse-matrix gap. *HPCC Week*, pages 9–10, December 1998.

[2] E. Arge, A. M. Bruaset, and H. P. Langtangen. Object-oriented numerics. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, pages 7–26. Birkhäuser, 1997.

[3] C. Ashcraft and R. G. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[4] C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *ACM Transactions on Mathematical Software*, 1999. To appear.

[5] S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc.

[6] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, 23(1):50–80, March 1997.

[7] F. Dobrian, G. Kumfert, and A. Pothen. Object-oriented design for sparse direct solvers. In D. Caromel, R. R Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 207–214. Springer-Verlag, 1998.

[8] F. Dobrian, G. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. In A. M. Bruaset, H. P. Langtangen, and E. Quak, editors, *Modern Software Tools in Scientific Computing*. Springer-Verlag, 1999. To appear.

[9] I. Duff and J. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, September 1983.

[10] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981.

[11] G. Kumfert and A. Pothen. An object-oriented collection of minimum degree algorithms. In D. Caromel, R. R Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 95–106. Springer-Verlag, 1998.

[12] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, January 1990.

[13] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, March 1992.

[14] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, September 1993.

[15] R. Schreiber. Scalability of sparse direct solvers. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56, pages 191–209. Springer-Verlag, 1993.