Oblio: A Sparse Direct Solver Library for Serial and Parallel Computations

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

As shown in [1], computing the solution of sparse linear systems of equations represents the key computation in several critical industry and national-security applications. Among the problems identified are the optimization of aircraft turbines, the design of public-key cryptosystems, and a broad range of other applications in computational fluid dynamics, linear programming, finite-element methods and process engineering.

We currently investigate sparse direct solvers, which are more robust than their iterative counterparts. The drawback is that they require more computational resources and they are more difficult to program. In terms of resources, decomposing a sparse matrix requires more work and storage than performing few steps of sparse matrix-vector multiply. In terms of programming, sparse direct solvers use sophisticated data structures and algorithms, including irregular and dynamic computations. However, while resource requirements are well understood, there is significant room for improvements from the programming perspective.

The major objective of this research is to build a modern software package that uses state-of-the-art sparse direct solver algorithms. The motivation for this work is twofold. First, research needs better tools for experimentation. It should be easy to combine algorithms in various ways as well as to prototype new ones. Second, real world applications need software that is easier to adapt to various architectural and application requirements. 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. From the application perspective, the systems to be solved may be unsymmetric, structurally symmetric or fully symmetric, real-valued or complex-valued, and pivoting may or may not be required for numerical stability.

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++. Performance reasons required certain tradeoffs and writing few low level computationally intensive kernels in Fortran 77. Currently, **Oblio** handles fully symmetric systems, both real-valued and complex-valued, and both positive definite and indefinite. In our opinion, the code is fairly easy to understand, modify and extend. As a result, it is potentially suited for experimentation and to match different architectural and application requirements.

Our work is part of the current efforts to introduce advanced software design to the scientific computing community. Most of the time advanced translates to object-orientedness, but other paradigms, such as generic programming, are also used. Currently, iterative solvers are the main targets for the new software techniques, PETSc (written in C) ([10]) and Diffpack (written in C++) ([4]) being well known examples. Less has been done to improve the design of direct solvers. **Oblio** 's sister, Spindle ([5]), 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 ([2]). 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 dissapear in the future.

We have presented an earlier, serial only, version of **Oblio** in [5]. In this paper we discuss the current version, highlighting modifications and extentions required by parallelism. In Section 2 we provide an overview of the problem, we address design issues in Section 3 and in Section 4 we report recent results obtained with the parallel code.

2 Problem Overview

Direct methods use a simple idea to compute the solution of a general linear system of equations $Ax = b$: decompose it into a sequence of systems which are easier to solve. This is achieved by splitting 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.

While conceptually simple, this strategy involves sophisticated computations when the system is large and sparse, a common case in a large number of applications. Solving a large system of equations requires a significant amount of computational resources and clever algorithms must be used to reduce this amount by taking advantage of sparsity. Storage and work are needed only for the nonzero entries in the coefficient matrix and factors and the sparsity of the factors depends on the row and column order in the coefficient matrix.

When the coefficient matrix is symmetric, it is usually decomposed as following:

$$PAP^T = LDL^T.$$

Here P is a permutation matrix that reorders A to preserve sparsity, and L and D are triangular and diagonal shaped factors. Note that, in order to save half of the storage and work, symmetry is also preserved by reordering rows and columns in the same way. Figure 1 shows a black box representation of the computation. The system's solution and right hand side are denoted by x and b respectively. A good place to start looking at sparse symmetric direct solvers is [7].

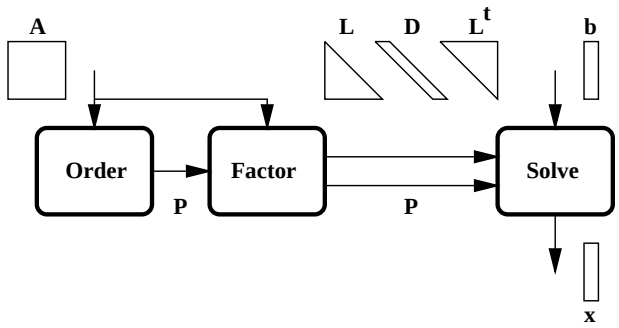

Fig. 1. A black box representation of a sparse direct solver.

In this paper we focus on factorization, a representative computational step and a good candidate for parallelization. In the remaining of this section we review major aspects concerning factorization algorithms. To simplify the discussion we assume that the coefficient matrix is already ordered to preserve sparsity.

As it turns out, sparse matrix computations can be illustrated in terms of graphs. An undirected graph $G(A)$ can be naturally associated with a symmetric matrix A: for each diagonal entry add a node; the node corresponds to the row and column in which the diagonal entry lies; for each pair of nonzero off-diagonal entries add an edge between the nodes corresponding to the rows and columns in which the nonzero entry pair lies. Figure 2 shows a sparse symmetric matrix and its associated undirected graph.

The factorization of A can be equivalently formulated as the procedure of eliminating its columns from left to right. Each time a column is eliminated it also updates the remaining ones. New nonzero entries, called *fill* entries are usually created by the update operations. Using $G(A)$, the factorization can be described as following: when a column is eliminated, delete its corresponding node and all its incident edges; also, add new edges between its remaining neighbors, if they

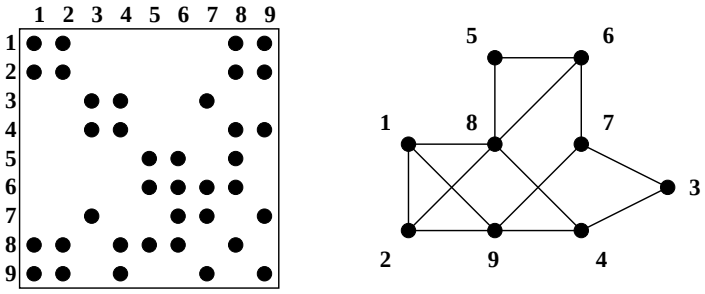

Fig. 2. A sparse symmetric matrix and its associated graph.

are not already connected. The new edges correspond to the fill entries and, consequently, they are called fill edges.

The graph $G^+(A)$ obtained from $G(A)$ by adding all the fill edges is called the fill graph and it corresponds to the fill matrix $F = L + D + L^T$ in the same way $G(A)$ corresponds to A. Thus, the amount of storage and work required to compute the factors is determined by $G^+(A)$. Figure 3 shows the fill matrix and the fill graph that correspond to the coefficient matrix from Fig. 2.

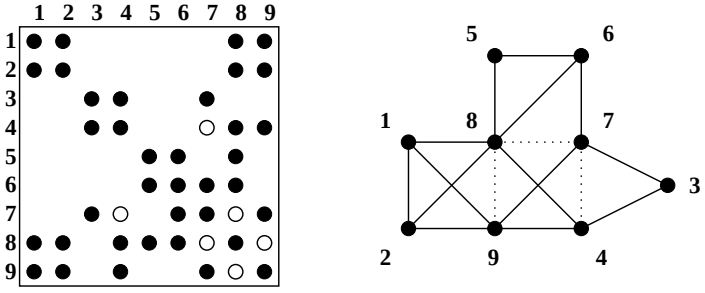

Fig. 3. The fill matrix and the fill graph.

Clearly, the factorization needs to be performed in the order in which the columns appear in the coefficient matrix, but there are reasons to organize it in a clever way. First, sparsity leads to irregular computations, which are not efficient. A significant increase in performance can be obtained by clustering columns to perform regular computations. Second, there is no total order among the factor columns, as if would be the case if A were dense. This represents a good oportunity for parallel computations.

Given $G^+(A)$, it is difficult to visualize how to organize the work to achieve better performance and to exploit parallelism. The key lies with a data structure that represents the backbone of the fill graph: the *elimination forest*.

Simply stated, the elimination forest $EF(A)$ is nothing but the transitive reduction of the fill graph. Such a definition is not correct though since the fill graph is not directed. This detail can be easily overcome by temporarily assigning an orientation to each edge. The direction is from its lower numbered to its higher numbered endpoint. The transitive reduction is computed for the resulting directed graph, edges in the elimination forest pointing from children to parent nodes. Edge orientation can then be dropped, from both $G^+(A)$ and $EF(A)$. For details about elimination forests see [8].

Here is how a factorization algorithm can use the elimination forest: it traverses it in topological order, usually in postorder; at each node it performs column elimination and update operations. The forest is very useful to define criteria for increasing performance and exploiting parallelism. The former is achieved by clustering columns along forest branches, where they have the identic or similar sparsity patterns. For the latter, independent branches must be processed in parallel.

By clustering columns the corresponding nodes are merged and the elimination forest is compressed. Nodes in the compressed forest correspond now to clusters of columns, also called *fronts*. The factorization proceeds in the same way, traversing the forest in postorder and performing some basic processing at each node. However, there is less overhead for fewer nodes, and the nodes are processed more efficiently, front computations being actually dense matrix computations.

Within this framework, choices can be made to organize the work. In particular, we use the *multifrontal* strategy, first mentioned in [6]. A good tutorial on multifrontal factorization in given in [9]. The procedure is easy to understand. The processing of a node begins by forming a dense matrix called the *frontal matrix*. Two types of columns are included in the frontal matrix: frontal columns, which belong to the current front, and update columns, which are columns that are updated by the frontal ones and which belong to a front that corresponds to an ancestor of the current node. Note the following key observation: the frontal columns are fully assembled (no other columns will update them) and so they can be eliminated; on the other hand, update columns are only partially assembled and their elimination must be postponed. Consequently, we only need to perform partial factorization on a frontal matrix and to propagate the updates upper in the forest.

To simplify the discussion we consider positive definite systems only. Consequently, there is no need to pivot for numerical stability and the processing of a frontal matrix is simply a partial Cholesky factorization.

Until now, the scheme we described it actually not particular to the multifrontal strategy. What makes the multifrontal factorization different is the way in which updates are propagated upper in the forest. Clearly, update operations can be performed between the current node and all its ancestors whose corresponding fronts include current update columns. The multifrontal strategy follows a different approach: updates are propagated from child to parent nodes. While not all the current update columns belong to the front corresponding to

the parent, they form a subset of all the columns of the parent's frontal matrix (frontal and update). This way updates are carried and aggregated along forest paths until they reach their final destination.

A second dense matrix, called the *update matrix*, is used to propagate updates between child and parent nodes. The update matrix is formed by extracting the update columns from the frontal matrix after performing the partial factorization. Since a parent node is processed only after all its children, a mechanism is needed to temporarily store update matrices. The natural solution imposed by the postorder traversal of the forest is an *update stack*, a stack of update matrices. Figure 4 shows the elimination forest that corresponds to the coefficient matrix from Figure 2 and the frontal and update matrices associated with the top nodes. Note how several nodes are clustered.

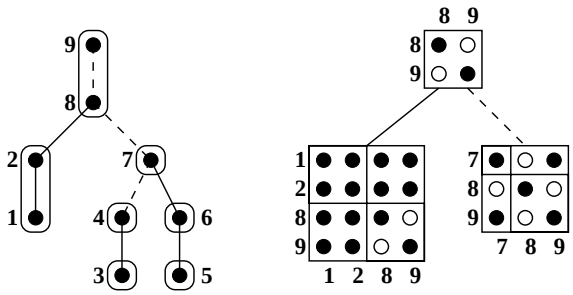

Fig. 4. The elimination forest, and the frontal and update matrices.

This is the basic mechanism of the multifrontal factorization. The rest is just details. For example, what if we solve more general indefinite systems? In this case a frontal matrix must be processed in a slightly different way. Pivoting needs to be performed within fronts and pivot selection criteria may actually delay the elimination of certain columns from one front to another one. In this situation update matrices must carry fully assembled columns in addition to partially assembled ones, as they migrate between fronts to increase their chance of becoming suitable pivots. Details on pivoting can be found in [3].

We conclude this overview by discussing parallelism. We have already noticed that independent branches can be processed in parallel. This is usually referred to as branch level parallelism. There is also front level parallelism: within each frontal matrix, as soon as a column is eliminated, all the frontal matrix columns to its right (both frontal and update columns) can be updated in parallel. Exploiting front level parallelism too is important because the higher layers of an elimination forest tend to have fewer independent branches and larger fronts.

The commom 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 partition, 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

Oblio was initially designed as a serial library. Recent modifications and extensions support parallel computations as well. Several components of the earlier version had to be adapted to the new requirements as parallel code must distribute both computation and storage.

We distribute cost dominant data structures but replicate few others. The arithmetic work is fully distributed. Basically, the same elimination forest guides both serial and parallel computations. In a parallel environment the forest is replicated and each processor traverses it in postorder. Additional mapping information helps processors skip nodes whose processing they are not responsible for (branch level parallelism) or partially process nodes (front level parallelism). Permutation objects are replicated too. We distribute coefficient matrix, factor, frontal and update matrix, and update stack objects.

This section is organized along six topics. We begin with a recent improvement: a class for array objects. We continue with classes that describe the objects visible at the highest level (coefficient matrix, permutations and factors), the elimination forest, and the factorization specific objects (frontal and update matrices, and the update stack). We pay special attention to the class that handles the factorization algorithm. In the end we compare the design of **Oblio** and SPOOLES.

3.1 Array Handling

Most **Oblio** objects store data in arrays. In the past we have used C++ arrays for this purpose but we recently introduced a new class to manage array objects. With C++ arrays certain statements are unnecessarily replicated throughout the code. More lines need to be typed and mistakes can be easily made. The `Array` template class (we store items of various data types), shown in Fig. 5, describes array objects in **Oblio**. The private data are placed in the top part and the public interface in the bottom part. We use similar figures throughout this section but be provide more details here in order to explain the principles we followed in the design.

For each class we usually have a regular constructor that builds non-empty objects, given object dimensions. We also provide default constructors because we sometimes need to build empty objects. The `resize` method can be used to change object dimensions.

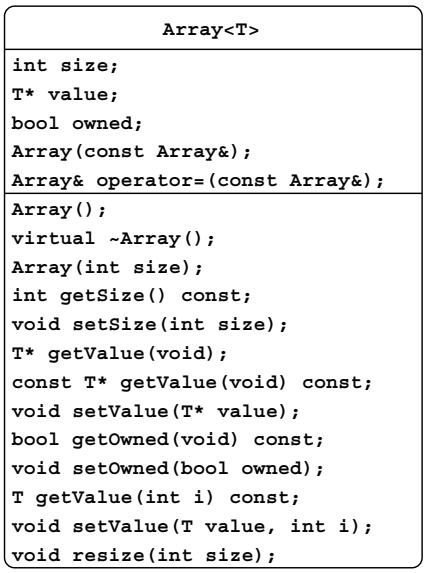

Fig. 5. A template class for arrays.

We currently avoid assignment operations between any of our objects as well as passing them by value. At present we do not find such operations necessary and, to prevent performance problems, we declare the copy constructor and the assignment operator private and we never define them.

We only allow reading object data fields, through get-like functions, except for `Array`, where set-like functions permit writing data fields. We also allow reading and writing array items.

A major difference between `Array` and other classes comes from the ownership flag. This flag is useful when we need to use externally allocated C++ arrays inside `Array` objects. In such situations we must have more control over the `Array` class: read and write any field. Note that this comes at a higher risk, so more attention needs to be paid to avoid potential mistakes when full control is exercised.

We do not allow the same degree of control for other classes since we do not find such an option useful. The fundamental operations allowed for instances of classes other than `Array` are: construction, destruction, resizing and querrying. Of course, their internal `Array` objects can be accessed according to the definition of the `Array` class.

We believe is is sometimes appropriate to have shared `Array` objects. This approach may bring significant storage savings when two different objects store identical data. While we do not provide such functionality yet, it would not be difficult to include it in a future version of the code. Clearly, a shared `Array` object must use reference counting to keep track of all the objects that use it.

3.2 Coefficient Matrices, Permutations and Factors

The objects visible at the highest level are coefficient matrices, permutations and factors. Mathematically they all represent matrices but the are certain differences between them that determine separate implementations. Currently, the corresponding **Oblio** classes are: `SparseMatrix`, `Permutation` and `SparseFactors`.

`SparseMatrix` is shown in Fig. 6. Because we solve both real-valued and complex-valued systems, `SparseMatrix` is a template class. The same is true for any other class that stores numerical data.

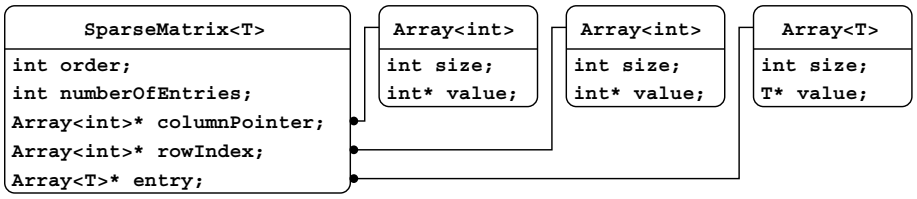

Fig. 6. The template class that describes coefficient matrices.

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, we prefer to store the whole coefficient matrix instead of saving half of the storage because this speeds up certain computations. This in one of several situations in which we trade storage for execution time. Note that instead of storing actual `Array` objects inside other objects we prefer to store pointers to them. This way it would be easier to add shared functionality later.

We omitted the `SparseMatrix` public interface to avoid too many details. An observation is necessary though. It applies to all classes described using `Array` objects. In the previous subsection we mentioned that except for the `Array` class we allow only reading private data fields. However, we do allow reading and writing items of the internal `Array` objects. This can be done in two ways. Naturally, one can retrieve the pointer to the `Array` object we are interested in and then use the methods of the `Array` class to access particular items. We also provide methods that access the items directly. The mechanism is basically the same but the querried pointer is not visible this time. There is a difference in performance between the two alternatives. The former is more efficient when a large number of items belonging to the same `Array` object must be accessed because a pointer to that object needs to be retrieved only once. Yet, the latter tends to make the code more readable.

`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 fill 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.

Figure 7 shows the `Permutation` class. The order is the single dimmension of `Permutation` objects. In addition, they store both direct and the inverse maps.

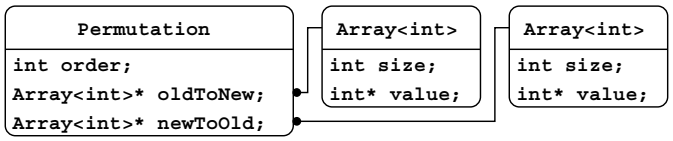

Fig. 7. A class for permutations.

Note that the inverse map can be easily computed from the direct one. Yet, we prefer again to trade storage for execution time. One potential problem with this approach is that, if we make the unlikely decision to discard the inverse map in a future version of **Oblio**, code that accesses the inverse map directly may break.

As we mentioned, `SparseMatrix` and `SparseFactors` objects are distributed when **Oblio** runs in a parallel environment, while `Permutation` objects are replicated. The distribution can be easily done with additional maps. For example a `SparseMatrix` object stored on a particular processor would then represent a fragment of a coefficient matrix.

3.3 The Elimination Forest

An elimination forest is described by the `EliminationForest` class. This specializes a the more general `Forest` class, from which it is derived. We use this scheme because we may need other forest-like objects in the future. Figure 8 shows both the `Forest` and `EliminationForest`classes. The dimension of any `Forest` object is represented by the number of nodes. Three `Array` objects store the parent, first child and next sibling for each node. An `EliminationForest` object adds another dimension: 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).

We also associate iterator classes with `Forest` because they provide a convenient mechanism to traverse `Forest` objects. At the end of the section we show how they are used by the multifrontal factorization.

Remember that an `EliminationForest` object is replicated by each processor when **Oblio** runs in a parallel environment.

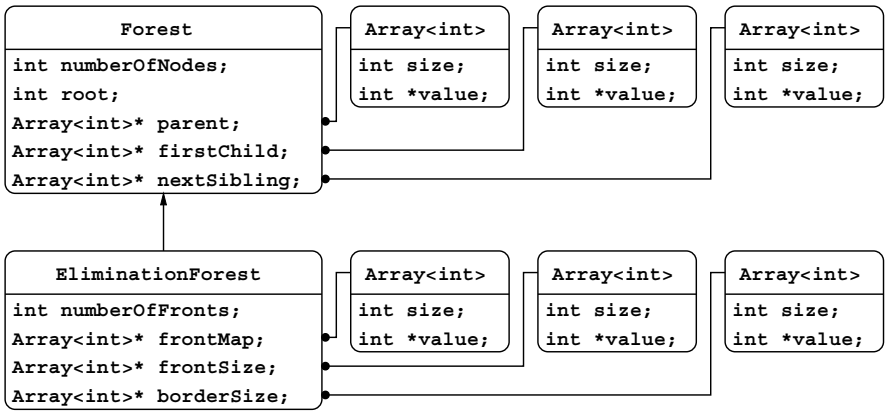

Fig. 8. The classes that handle forests and elimination forests.

3.4 Factorization Specifics

The postorder traversal of the 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 were one must focus on performance and exploit front level parallelism. Frontal and update matrices are the key elements during the processing of a front.

Both frontal and update matrices are dense. To capture their common description we first implemented a `DenseMatrix` class. From this we specialized `FrontalMatrix` and `UpdateMatrix`. Since this is the place where most of the difference between a serial and a parallel code come from, we discuss both the initial approach and the current solution.

The first solution we came up with in our serial code is shown in Fig. 9. The only dimension of a `DenseMatrix` object is the order. The entries are stored by an `Array` object. There are two additional `Array` objects. The first stores column pointers. While this information can always be computed from the order of the matrix and the column indices, we trade storage for execution time again. The second additional `Array` object stores a map from local indices (respective to the `DenseMatrix` object) to global indices (respective to the factors).

`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.

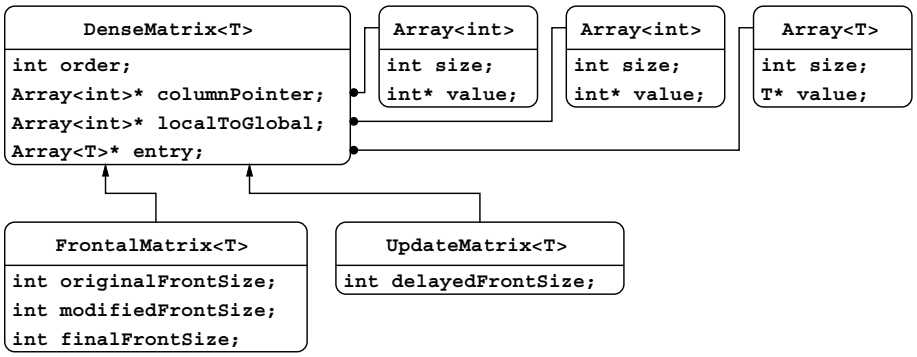

Fig. 9. Initial approach for frontal and update matrices.

`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 allocates 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. 10. 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.

Finally, `UpdateStack`, shown Fig. 11, describes the stack that stores update matrices during the factorization. The dimensions of an `UpdateStack` object are its maximum and its current size. One `Array` object stores pointers to the stacked update matrices. The basic functionality allows pushing and popping

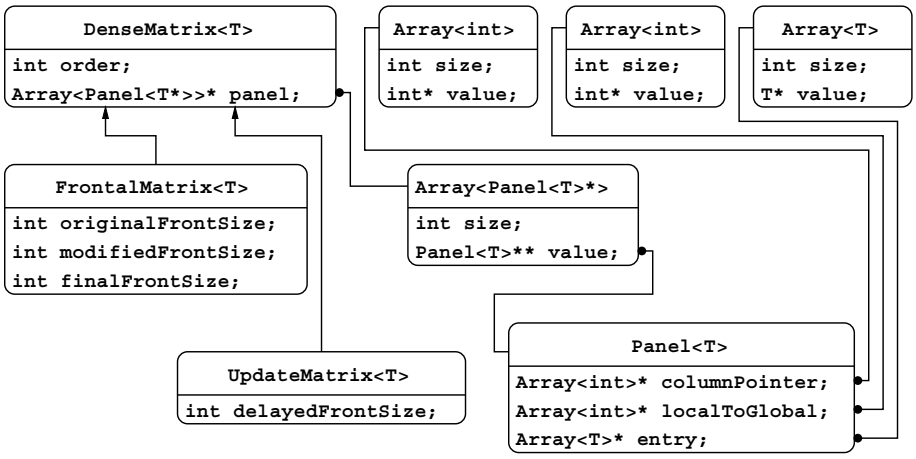

Fig. 10. Current solution for frontal and update matrices.

`UpdateMatrix` objects. In a parallel environment the distribution of the update stack is determined by the distribution of the update matrices.

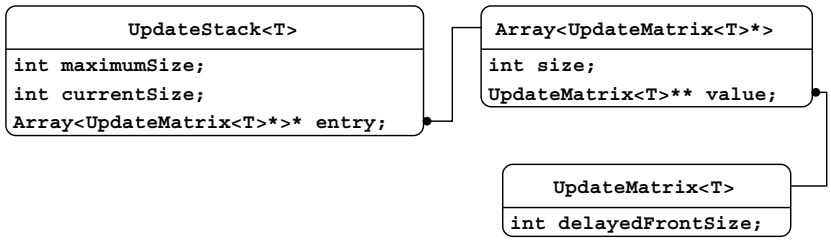

Fig. 11. The class that manages update stacks.

3.5 The Multifrontal Factorization

We end our discussion about design with a closer look at the factorization. There are several algorithms that we have not made methods inside classes. Instead, we designed separate classes for these algorithms. As a consequence, there are two types of classes in **Oblio**: *data* classes and *algorithm* classes. The distinction may be a little fuzzy because data classes have methods and algorithm classes have state information. `MultifrontalFactor`, which describes multifrontal factorization objects, is an example of an algorithm class.

One reason for this approach is the fact that it does not seem natural to associate a factorization algorithm with any other class. Should it be associated with `SparseMatrix` or with `SparseFactors`? Another reason is that each time a new type of factorization would be added (left-looking, right-looking), the class the algorithm is associated with would need to be modified.

Algorithm classes are described by state information and by methods that perform various tasks. The most representative method is `run`, which executes the particular algorithm. Figure 12 describes the core of the `run` method associated with `MultifrontalFactor`. We use a reduced version of the code here to keep the discussion simple. The actual code is more general because it handles pivoting too.

The computation represents the outer loop of the factorization, which traverses the elimination forest in postorder, using an iterator. At each step (node) a frontal matrix is created and initialized. Frontal matrix entries come from the coefficient matrix or from the update matrices corresponding to the children of the current node. Partial factorization is then performed on the frontal matrix and its entries are moved to the factors and to the update stack.

Several objects used in this piece of code are parts of the state information stored by each `MultifrontalFactor` object: a coefficient matrix (`a_`), a permutation (`p_`), a set of factors (`ldl_`), an elimination forest (`f_`) and an update stack (`u_`). In addition, `globalToLocal` is an `Array` object that translates from factor to frontal matrix indices.

3.6 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 forest data strucure first proves to be right, SPOOLES also using a specialized 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 approcahes slightly different. It is not clear to us which one is superior. Third, the factorization algorithm is associated with the data structure that describes the factor. In C++ this would be equivalent with making the factorization algorithm a method within class. As we already explained, we believe our approach has certain advantages.

```
// For every node in the elimination forest ...
for (EliminationForest::postorderIt it = f_->postorderBegin();
     it != f_->postorderEnd();
     it++)
{
  // Create frontal matrix.
  FrontalMatrix *fm = new FrontalMatrix(f_->frontSize(*it) +
                                        f_->borderSize(*it),
                                        f_->frontSize(*it),
                                        f_->frontSize(*it));
  fm->loadIndices(*f_, *it); // Load original indices.
  fm->computeGlobalToLocal(globalToLocal); // Compute translation map.
  fm->fill(0.0); // Zero out entries.
  fm->loadEntries(*a_, *p_, globalToLocal); // Load original entries.

  // For every child of the current node ...
  for (int k = f_->getNumberOfChildren(*it); k > 0; k--)
  {
    UpdateMatrix *um = u_->pop(); // Pop update matrix.
    fm->loadEntries(*um, globalToLocal); // Load update entries.
    delete um; // Destroy update matrix.
  }

  fm->factor(); // Perform partial factorization.

  fm->saveIndices(*ldl_); // Save eliminated indices.
  fm->saveEntries(*ldl_); // Save eliminated entries.
  // Create update matrix.
  UpdateMatrix *um = new UpdateMatrix(fm->getOrder() -
                                      fm->getFinalFrontSize(),
                                      fm->getModifiedFrontSize() -
                                      fm->getFinalFrontSize());
  fm->saveIndices(*um); // Save update indices.
  fm->saveEntries(*um); // Save update entries.
  delete fm; // Destroy frontal matrix.
  u_->push(um); // Push update matrix.
}
```

Fig. 12. The core of the multifrontal factorization algorithm.

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 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

Table 1. Results on an SGI Origin 2000 multiprocessor.

problem	order	time (s)	speedup			
		1 P	2 P	4 P	8 P	16 P
grid9.127	16,129	0.62	1.72	2.58	2.70	2.00
grid9.255	65,025	3.97	1.81	3.01	3.85	4.67
grid9.511	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
grid3d	29,791	53.96	1.69	3.09	5.38	8.15
grid3dt	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

time with performance tuning. Second, it is known that sparse direct solvers do not scale well ([11]). Communication cost, shown in Table 2 for both two-dimensional (2D) and three-dimensional (3D) problems, is a limiting factor. Basically, there are two mapping and two communication models. A one-dimensional (1D) mapping decomposes the computation only along columns, while a two-dimensional one decomposes the computation along both rows and columns. As for communication it may or may not be overlapped with computation.

Table 2. Communication cost.

	without overlap		with overlap	
	2D problem	3D problem	2D problem	3D problem
1D mapping	np	$n^{4/3}p$	n	$n^{4/3}$
2D mapping	$n\sqrt{p}$	$n^{4/3}\sqrt{p}$	$n/\sqrt{p}$	$n^{4/3}/\sqrt{p}$

As Table 2 suggests, 2D mapping must be preffered over 1D mapping and communication should be overlapped with computation. However, the analytical results are obtained with an oversimplified model which ignores several issues that are present in practice. We currently use the weaker 1D mapping because it simplifies pivoting and we do not yet overlap communication with computation.

Our next steps will include 2D mapping and overlapped communication as extentions to **Oblio**. Having both mapping and both communication models available within the same framework will be extremely useful because we plan to run experiments with hybrid algorithms. These algorithms switch mapping and communication models dynamically during the elimination forest traversal, as different fronts may have different requirements. A significant contribution to supporting such experiments comes from the object-oriented design.

References

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

[2] C. Ashcraft and R. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the SIAM conference on parallel processing for scientific computing*, 1999.

[3] C. Ashcraft, R. Grimes, and J. Lewis. Accurate symmetric indefinite linear equation solvers. To appear, 1999.

[4] A. M. Bruaset and H. P. Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, pages 50–80, 1997.

[5] F. Dobrian, G. Kumfert, and A. Pothen. The design of sparse direct solvers using object-oriented techniques. To appear, 1999.

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

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

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

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

[10] Balay S., Gropp W. D., Curfman McInnes L., and Smith B. F. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[11] R. Schreiber. Scalability of sparse direct solvers. In *Graph Theory and Sparse Matrix Computation*, pages 191–211. Springer-Verlag, 1993.