Performance and Portability Aspects of Object Oriented ILU Library Design with C++ Implementation

David Hysom and Alex Pothen

Old Dominion University, Norfolk VA 23529-0162 USA

Abstract. We discuss the design and implementation of a highly parallel, extensible algorithmic framework for ILU preconditioner computation. The framework permits selection of factorization algorithms on a subdomain by subdomain basis, in accordance with a problem's underlying physics. To enhance portability certain advanced C++ language features had to be abandoned. To increase performance we augmented the two-tiered interface/implementation design model with a mid-level interface, which makes use of *incomplete data encapsulation and abstraction* (IDEA). Preliminary results indicate highly scalable behavior, with good speedup.

1 Introduction

Although performance is frequently cited as the over-arching concern in scientific computing codes, portability is crucial to a software package's survival. Authorship of object-oriented C++ code that is both efficient and portable is problematical at best. Portability takes on new shades of meaning when designing an incomplete factorization (ILU) library.

An ILU preconditioner package is not, by its very nature, a stand-alone product. Unlike direct LU solvers, ILU functionality only makes sense as a module in a larger iterative solution procedure. An ILU library functions as an extension to an existing, *parent* library, e.g., PETSc [4]. The parent library may be written in some other language, such as C or Fortran, which, given the state of today's compilers, is ipso facto more portable than ANSI C++. Yet, an ILU library is useless if it does not compile and link on the platforms to which the parent library has been ported.

Hence, a decision to implement in C++ should not be taken lightly. We found that use of some of the most powerful C++ features, such as templates, the standard library, and the const and mutable keywords, resulted in codes which were elegant and understandable, but died a quick compile time death on many platforms. Why, then, do we still prefer C++ as our implementation language of choice?

Many numerical codes continue to be written in Fortran; these codes are highly efficient and portable, but tend to be monolithic rather than modular, and difficult to understand and modify. One alternative [1, 4] is the combination of object oriented design with implementation in C, and the use of Fortran kernels and/or the BLAS for numerically intense computations. This leads to code which is efficient, modular, and portable, but extensibility and ease of modification remain problems. Navigating class hierarchies and unraveling compositional relationships—which are typically hidden within structures—can be arduous. Implementation in C++ is attractive from an organizational standpoint since a class's inheritance graph, compositional structure, and public and private interfaces can be easily and clearly delineated in header files. C++ was also specifically designed to support polymorphism, dynamic binding, and data encapsulation, and other object-oriented concepts.

Having elected to implement in C++, portability issues precluded the use of several advanced language features, such as templates, exception handling, iterators, and use of the Standard Template Library (STL). We addressed performance issues by augmenting the two-tiered Object Oriented Design (OOD) approach, which separates abstraction from implementation, with a mid-level interface, in which direct manipulation of other classes' primitive data types is permitted. However, as explained in Section 3, these primitive data types are themselves a partial abstraction, and do not necessarily coincide with an object's private internal data representation. We call this concept *incomplete data encapsulation and abstraction* (IDEA).

2 Background

A preconditioner M is often used to transform a large, sparse linear system, $Ax = b$, into a related system possessing better convergence properties, $M^{-1}Ax = M^{-1}b$. Direct computation of the approximate inverse, M, can easily be carried out in parallel once M's sparsity pattern is given. Unfortunately, determination of sparsity patterns which result in highly effective preconditioners has remained somewhat elusive.

Incomplete factorization (ILU) preconditioning is based on the computation of triangular factors L and U, where $LU \approx A$. Preconditioner application reduces to the solution of two triangular systems, $Ly = b$ and $Ux = y$, during each iteration. ILU factors can be structurally based and computed using graph search algorithms (ILU(k)), or numerically based, in which case they are computed using a drop-tolerance threshold (ILUT). Partial pivoting may be used with either method for indefinite problems. ILU preconditioning is widely used in serial contexts and recognized as robust and effective, however, computation of the LU factors is widely held to be primarily serial in nature.

We have developed and implemented a parallel algorithmic framework that supports computation of all ILU variants. We begin with a matrix whose underlying graph has been partitioned into subdomains, such that each subdomain has a large ratio of interior to boundary vertices. For ease of presentation, we will assume a one-to-one mapping between subdomains and processors. This sort of partitioning frequently arises during the numerical solution of discretized Partial Differential Equations.

A *processor graph* is a graph whose nodes correspond to subgraphs (subdomains) of A, and whose edgeset contains edge (r, s) if subgraphs r and s are joined by one or more edges. Given a partitioned, distributed matrix, a *processor graph* is easily formed by scanning through a matrix's adjacency lists. The resulting graph is colored to identify independent sets, and an ordering is imposed on the sets. We impose the constraint that the processor graph of the factor $F = L + U - I$ and the matrix A be identical. This permits the processor graph to be used to identify all communication patterns prior to the factorization phases; it also preserves parallelism during the triangular solves.

Within each subdomain, interior nodes are ordered before boundary nodes; by the fill path theorem [3], this ensures that fill edges joining subdomains can only exist between boundary nodes. This permits independent factorization of subdomain interiors, and thus allows selection of ILU algorithms and parameters (e.g, level, for ILU(k)) on a subdomain by subdomain basis. Hence, the framework is ideally suited for adaptive preconditioning in multi-physics problems, wherein different models are used for different physical subdomains.

Figure 1 presents an overview of our parallel approach, as applied to a symmetric matrix with interleaved symbolic and numeric factorization.

- Input: well-partitioned, distributed matrix
- Form processor graph and order vertices to reduce directed path lengths (global graph coloring)
- Locally reorder interior, then boundary nodes
- Factor interior rows
- Receive row-information from lower-ordered neighbors in processor graph
- Factor Boundary Nodes
- Send row-information for boundary nodes to higher-ordered neighbors in processor graph

Fig. 1. Parallel Algorithmic Framework

3 Library Design

OOD guidelines advise that one should define an *abstract model* that is independent of any programming language or implementation. The abstract model is defined by its public interface, which specifies the operations which may be performed on class objects, but makes no mention of how they are to be performed, or of the class's private data representation. This separation ensures that modifications made to private implementations and data will neither break nor introduce errors into application codes which use the interface.

Although well established and of proven utility, this two-tiered approach can result in poor performance. One of the most common operations in scientific code is the traversal of the nonzero elements of a matrix row or column. In traditional C and Fortran implementations, a row's nonzero elements are stored in a contiguous section of an array, and the traversal is efficiently performed using pointers or indexing. It is possible to encapsulate the data and perform such traversals using iterators or other methods. However, given the state of today's compilers, these methods usually result in code that executes much less efficiently than does code which directly accesses the underlying, primitive data structures.

The concept of IDEA addresses this deficiency. The methods specified in a mid-level interface return pointers to primitive data types, i.e, arrays of integers and doubles. This permits non-member methods to efficiently manipulate data. In the next section we discuss the specifics of this data access; we then summarize the prominent features of each of our library's three levels.

3.1 Access to Private Data

We distinguish between *public* and *mitigated* access to private data. Public access, which we do not support, permits any function, method, or object to gain unilateral read/write access to an object's data. Mitigated access, in contrast, permits the direct manipulation of primitive data types, but only after access is gained through the public interface.

Mitigated access preserves the separation of abstraction from representation. For example, the method `double* Vector::borrowBegin();` is guaranteed to return a pointer to an array of doubles, but says nothing about the Vector's private representation of the data. Internally, the Vector's data may be stored in any of a variety of formats, e.g, a linked list with the double array being assembled on-the-fly when the call is invoked, or encapsulated within a shared array object. The private implementation remains undefined by the public interface. Thus, when we speak of a Vector's double array, we are referring to a data structure that is defined at the abstract level.

3.2 Mid-level Interface

At the abstract level, our Matrix class employs the standard compressed row storage scheme for sparse matrices. This scheme consists of three arrays, `rp`, `cval`, and `aval`. The values in the `rp` (row pointer) array indicate where the column numbers and nonzero values for each row begin in the `cval` and `aval` arrays. Thus, in a uniprocessor environment, the ith row has nonzeros in columns `cval[rp[i]]`, `cval[rp[i]+1]`, ..., `cval[rp[i+1]-1]`, and the nonzero values are stored in corresponding locations in the `aval` array. We also maintain a `diag` array, which indicates where the diagonal elements in the `cval` and `aval` arrays are located. In parallel environments Matrix instances are distributed, with each processor maintaining a unique section of contiguously numbered rows, and the storage scheme is modified by the addition of a scalar `offset`. Thus, the ith row's nonzeros are stored beginning at `cval[rp[i]+offset]`.

We allow mitigated access to our Permutation's integer arrays, our Vector's double array, and the arrays comprising a Matrix's sparse storage representation.

Users are required to notify an object when they are finished examining or modifying the array data. A prototypical definition of a mid-level interface follows.

```
Class Vector
{
  public:
    const double* borrowBegin() const;
    void          borrowEnd() const;
    double*       borrowForUpdateBegin();
    void          borrowForUpdateEnd();
    ...
};
```

The first pair of calls manages read-only access, and the second manages read/write access. Since private data representations may differ from those specified in the abstract mid-level model, the `borrowForUpdateEnd()` call is necessary so that the object can incorporate any changes made during read/write into its internal data representation. Additionally, the matched pairs of calls permit for robust error checking, e.g, an array should not be simultaneously borrowed in read and read/write modes.

A mid-level interface may be specified to enable access to data in a variety of forms. For example, a Matrix class could include both `borrowBegin()` methods, which return arrays representing the matrix in compressed row storage, and `borrowBeginDense()` methods, which return dense representations.

3.3 High Level Abstract Model

Figure 2 contains a simplified diagram of our Factorization class hierarchy, and Fig. 3 shows Matrix class composition. All classes inherit from an Object class (not shown in the diagrams), whose primary function is error management. The existence of a common progenitor eases the task of altering error handling to conform to different parent libraries' expectations.

Communication processing is encapsulated in the NonLocalSend and NonLocalReceive classes. There is actually a hierarchy of NonLocal classes, since communication requirements vary depending on whether the matrix is symmetric or unsymmetric, and on whether the factorization being performed is symbolic, numeric, or, as in the case of ILUT, both.

Instantiated Matrix and subservient class objects are fully distributed. For example, a Matrix's Permutation object contains only necessary data, i.e, permutation information for the vertices (rows) in the processor's subdomain.

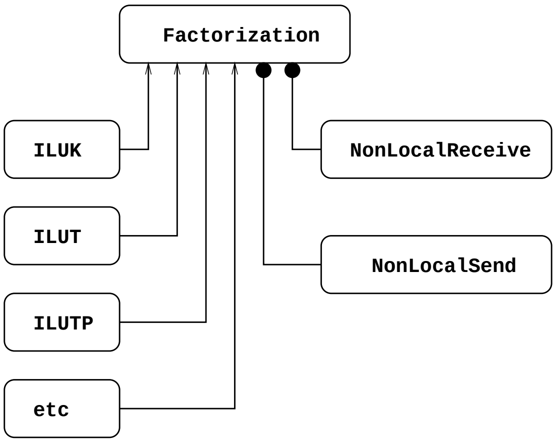

Fig. 2. Factorization class hierarchy and composition

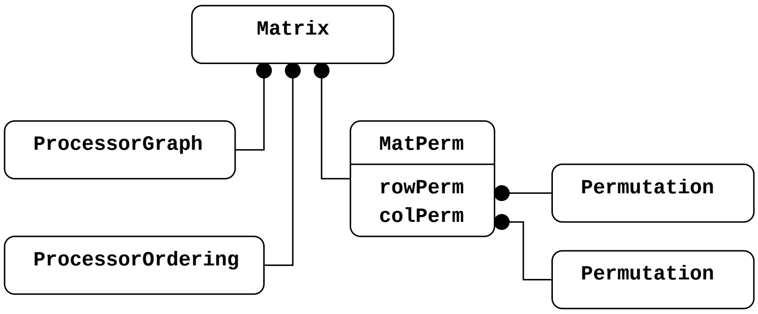

Fig. 3. Matrix class composition

Some parent libraries fully define a user interface, in which case the details of our public interfaces may be entirely hidden from end users. Using the library with other packages requires slightly more involvement on the part of the user. Figure 4 illustrates how methods in our mid-level model are used to interface with a parent library, while Fig. 5 shows how the high-level interface is used to access library functionality.

3.4 Array Implementation: Shared Objects

Given that A is a large matrix and data duplication should be minimized, an important implementation issue concerns appropriate ways of handling copy operations such as

```
Matrix A("filename");
Matrix B = Matrix A;
```

```
//PETSc library calls
    Vec petsc_x;
    Scalar *a;
    VecGetArray(petsc_x,&a);
    int size = VecGetSize(petsc_x);
//ILU mid-level library calls
    Vector X(size);
    int *x = X.borrowForUpdateBegin();
    memcpy(x, a, size*sizeof(double));
    x.borrowForUpdateEnd();
```

Fig. 4. Typical usage of mid-level model to interface with parent library

```
//Preconditioner Setup Phase:
    Matrix A(filename);
    FactorILUK sf;
    Options opt;
    opt.level = 3;
    sf.setOptions(opt);
    Matrix SF = sf.run(A);        //perform symbolic factorization
    FactorNumeric nf;
    Matrix F = nf.run(A, SF);     //perform numeric factorization
```

Fig. 5. Sample code for accessing ILU library functionality

A common practice is to employ a shared data scheme wherein objects maintain ownership flags, and the array's owner is tasked with memory management responsibilities [1, 2]. Correct handling of ownership flags, however, can become quite tedious when dynamically allocated objects, scoping issues, and read/write behavior come into play.

We address this issue by encapsulating integer and double arrays in shared Array classes, with copy-on-write capabilities. Making class destructors private ensures that the Arrays can only be instantiated on the heap, which resolves scoping issues. After an object acquires a pointer to an Array held by some other object, our protocol requires it to invoke the call, `array.hello()` and, when it is completed with the Array, in place of the destructor it should invoke the call `array.goodbye()`.

The Arrays maintain internal usage flags, which are incremented or decremented by various calls. In a very real sense the Arrays "own themselves." When an Array object's usage count reaches zero, it suicidally invokes its own destructor.

Since almost all of these details are hidden in class constructors, destructors and assignment operators, these details are primarily of interest to class developers, rather than end users.

4 Results

For testing purposes we use internally generated matrices corresponding to two-dimensional, five-point stencil grids. Two strategies are used to partition matrix rows amongst processors. With respect to the underlying grid, **block partitioning** divides the grid into square sub-grids, while **striped partitioning** divides the grid into horizontal slices. Each block or slice constitutes a subdomain, and there is a one-to-one mapping between subdomains and processors.

Figure 6 shows efficiency for combined symbolic and numeric ILU(k) on an SGI Origin2000. Efficiency is seen to increase with level, and is 50% or above for all trials. Table 1 compares efficiency for blocked vs. striped partitioning. Efficiency for the striped cases is inferior to that for the blocked cases, which is expected, since the interior/boundary ratio is less optimal with striped partitioning However, these results indicate that our methods can produce favorable results even for matrices corresponding to naturally ordered grids.

We are currently profiling and tuning our code, and constructing an interface to PETSc. When this is done we will be able to report statistics for the complete iterative solution of linear systems. We are also investigating ways of reducing communication costs.

4.1 Performance and Scalability Analysis

To introduce performance and scalability behavior, we consider an ILU(k) factorization on an $N \times N$ matrix corresponding to a 2D, five-point stencil grid. The matrix is assumed partitioned amongst p processors, with each processor's interior nodes naturally ordered. In keeping with real-world architectural and problem parameters, we assume a large interior to boundary node ratio, and a relatively small level; mathematically, $p \ll N$ and $k \ll \sqrt{N/p}$.

Asymptotically it can be shown that, following ILU(k) factorization, for $k \geq 1$, each lower triangular row in the factor contains at most $2k$ non-zero entries, and factoring a row entails $O(k^2)$ work. Assuming a non-contentious communication model wherein α and β represent message startup and cost-per-word respectively, and with all arithmetic operations normalized to unity, speedup can be expressed as

$$\text{speedup} = \frac{4k^2N}{\frac{4k^2N}{p} + 4(\alpha + 4\beta k\sqrt{\frac{N}{p}}) + 4k^2\sqrt{\frac{N}{p}}}$$

The three terms in the denominator are the costs for the elimination of interior nodes, the communication of boundary row information, and the elimination of boundary nodes. For a fixed problem size and number of processors, the computational cost (first and third terms) is proportional to k^2, while the communication cost (second term) is proportional to k. Hence, efficiency is expected to increase with level; this is evident in the experimental results presented above.

Assuming p is a perfect square, on regular 2D grids the asymptotic ratio of interior to boundary nodes is $\sqrt{N}/(4\sqrt{p})$. Recalling the constraint that subdomains have large interior to boundary node ratios, we are interested in behavior as we increase problem size by an arbitrary factor x, while keeping k and N/p constant. For a block partitioned grid, the interior/boundary node ratio remains constant. Hence, our simplified model predicts perfect scalability. These results extend to meshes, embedded in two or three dimensions, that have good separators.

5 Summary

The two-tiered OOD model, which separates class abstraction from implementation, and mandates data encapsulation, is at odds with the the requirement that scientific codes be both portable and efficient. This state of affairs is at least partially attributable to a disparity between compiler implementations and the ANSI C++ Standard. In order to improve performance and portability we have formally introduced a third level in the object-oriented design model, which maintains abstraction while permitting direct access to primitive data structures.

Although our presentation has centered around the design of an ILU library, we believe that a three-tiered OOD model has broad applicability for scientific code development. Many researchers continue to prefer Fortran and C for application programming; hence, object-oriented C++ libraries should provide APIs for both these languages. BPKIT [2], for example, is implemented in C++ but provides code for accessing library functionality from both C and Fortran applications. This sort of language interoperability can best be accomplished by accessing and passing primitive data structures, e.g, scalars and arrays of integers and floats. Our concept of a mid-level level interface is ideally suited for this purpose.

Table 1. Performance comparison between blocked and striped partitioning strategies for a 2D 480 x 480 grid; data from SGI Origin2000.

	Efficiencies (%)			
	p = 4		p = 16	
level	blocked	striped	blocked	striped
0	51	57	22	21
1	58	56	26	23
2	69	65	33	31
4	79	76	45	43
6	83	78	62	54
8	88	77	57	53
10	93	82	68	66
12	92	76	78	70

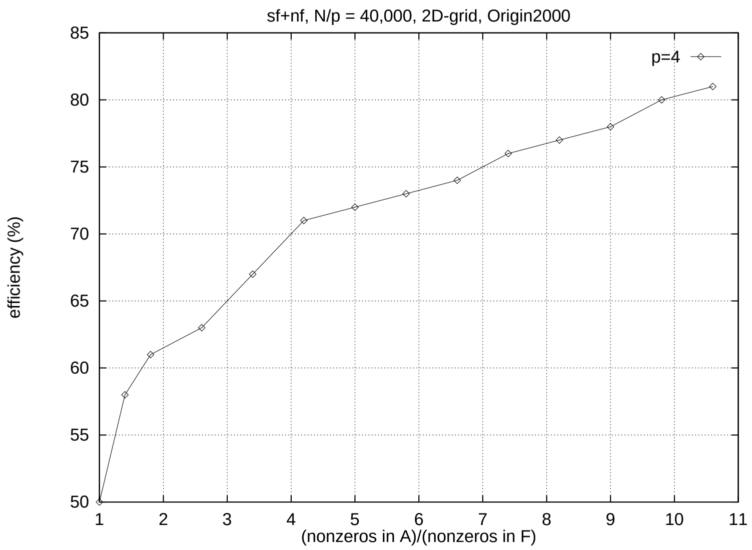

Fig. 6. Performance for ILU(k), $k = 0 \ldots 12$ for block-partitioned 2D grid, $N/p = 40,000$ on an SGI Origin2000

References

[1] C. Ashcraft, D. Pierce, D. K. Wah, and J. Wu. *The Reference Manual for SPOOLES, Release 2.2*, February 1999.

[2] E. Chow and M. A. Heroux. Block preconditioning toolkit. http://www.mcs.anl.gov/petsc, 1997.

[3] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.*, 23(1):176–197, 1978.

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