On the Design of Interfaces to Sparse Direct Solvers

MARZIO SALA
Eidgenössische Technische Hochschule Zürich
KENDALL S. STANLEY
Oberlin College
and MICHAEL A. HEROUX
Sandia National Laboratories

We discuss the design of general, flexible, consistent, reusable and efficient interfaces to software libraries for the direct solution of systems of linear equations on both serial and distributed memory architectures. We introduce a set of abstract classes to access the linear system matrix elements and their distribution, access vector elements, and control the solution of the linear system.

We describe a concrete implementation of the proposed interfaces, and report examples of applications and numerical results showing that the overhead induced by the object-oriented design is negligible under typical conditions of usage. We include examples of applications, and we comment on the advantages and limitations of the design.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Parallel Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques–*Object-oriented design methods*; D.2.13 [**Software Engineering**]: Reusable Software–*Reusable libraries*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra–*Sparse, structured, and very large systems (direct and iterative methods)*

Additional Key Words and Phrases: Direct Solver Libraries, Object-Oriented Design, Distributed Linear Algebra.

1. MOTIVATIONS

This paper describes the design and the implementation of interfaces to libraries for the serial and parallel direct solution of linear systems of type

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a real and sparse square matrix, and $x, b \in \mathbb{R}^n$ are the solution and the right-hand side vectors, respectively.

Authors' address:
Marzio Sala: Department of Computer Science, ETH Zürich, CH-8092 Zürich, Switzerland.
Ken Stanley: Department of Computer Science, Oberlin College, Oberlin, Ohio, USA.
Michael A. Heroux: PO Box 5800 MS 1110, Albuquerque, NM 87185-1110, U.S.A.
ASCI program and the DOE Office of Science MICS program at Sandia National Laboratory. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Generally speaking, a direct solution algorithm for (1) is any technique that computes three matrices, L, D and U, such that $PAQ = LDU$, where P and Q are permutation matrices and the linear systems with matrices L, D and U are easy to solve [Golub and Van Loan 1996]. The process of computing these three matrices is called *factorization*. Typically, L is a lower triangular matrix, U is an upper triangular matrix, D is a diagonal matrix (or possibly the identity matrix), and the algorithm adopted for their computation is some variant of the Gaussian elimination method.

Direct sparse solvers are an essential kernel in scientific and engineering computing and are required by several numerical algorithms, even within the same application. An incomplete list would include implicit time-stepping schemes, Newton-like methods, multilevel and domain decomposition preconditioners. In some instances these solvers are applied to the original problem in (1). Increasingly these solvers are used as a phase of a more complex solution process where the direct solver is used as a coarse grid problem solver, or is used to solve a set of subproblems, such as in FETI [Farhat and Roux 1992] methods. Thus, even though the parallel scalability of direct sparse solvers is usually limited to a factor of 10-100 speedup, these solvers remain a critical technology for large-scale simulations.

The development and implementation of direct solvers for (1) is a challenging task that has been a subject of research over the last four decades. A brief overview of direct solution algorithms is given in Section 2; however, the focus of this article is on the *usage* of direct solution methods, rather than on their development or analysis. Therefore, the point of view is the one of users (that is, application developers), that are interested in solving (1) using already available software libraries.

In contrast with dense solvers, where one can take advantage of the widely-available suite of solvers contained in the LAPACK library [Anderson et al. 1999] and ScaLAPACK library [Blackford et al. 1997] or the PLAPACK library [Alpatov et al. 1997], there is no single direct solver library of choice for (distributed) sparse linear systems. Since no "gold standard" exists, application developers aiming to solve (1) generally have to look for a suitable library that meets their needs. This search is affected by the following questions:

(Q1) Software ease-of-use and availability: Is the software library easy to use, stable, well-documented and actively supported?

(Q2) Special numerical properties: Does the solver support the special properties of my problems? For example, does the solver support symmetric matrices, reuse of matrix patterns, robust pivoting, etc.?

(Q3) Time-to-solution: Is the solver efficient in terms of CPU time?

(Q4) Memory use: How much storage does the solver require?

The relative importance of these aspects is usually subjective. For relatively simple problems, (Q3) is usually the key point, followed by (Q4). For complicated problems, and especially in industrial applications, (Q2) is of paramount importance. Point (Q1) is usually considered as "technical details" of minor importance, even if they aren't. At this time there are many viable direct sparse solvers, each of which has a special "niche" in the user community because it addresses the above questions appropriately for some users.

Using libraries can be a non-trivial task, since library developers often include a wide variety of choices, aiming to provide a powerful and comprehensive library. Furthermore, many solver parameters are *ad hoc* and users have little *a priori* sense of optimal parameter values. Application developers, in contrast, are usually not experts in direct sparse solvers, and do not aim to become such. They often choose the code that looks easier to use and seems to perform in a reasonable way. This difference of views creates a gap between the actual and potential use of mathematical software. Of course, a simple but inefficient library is of no interest; therefore the "optimal" library will exhibit a mixture of the features outlined above.

Once a library has been chosen, the application developers have to write a custom-made interface between the selected library and the application. This usually requires storing the linear system matrix A using the storage format required by the library, then calling the correct sequence of instructions to factor the matrix and solve the linear system. In our opinion, the approach is sub-optimal for both application and library developers because it:

(1) *Offers partial coverage:* Writing a custom-made interface means that only the targeted library will be used. This is inconvenient because of the already mentioned difficulty to choose *a-priori* the best library for a given application. In some cases, a theoretical analysis of the problem at hand can suggest the right algorithm. Alternatively, one can consider numerical comparisons on test matrices available in the literature, see for instance [Amestoy et al. 2001; Gupta 2001] and the references therein. Often, however, one has to validate a given library on the application, architecture, and data set of interest, and this can be done only if the interface is already available;

(2) *Produces maintenance problems:* Including the interfaces within the application code requires the application developers to manipulate the matrix format, the memory management, the calling conventions, and others, that can vary from one library to the following. Although not necessary difficult, these activities are usually time-consuming;

(3) *Delays the usage of new libraries:* Large application codes usually have a long life. The sheer size of the codes and a natural reluctance to change successful projects discourage any effort of rewriting unless absolutely necessary. Since a new library (or a new version of a given library) may require a new matrix format or distribution, or new calling conventions, application developers may simply decide to continue using the interface already developed.

This article shows that it is possible to address these problems by using object-oriented (OO) design and programming. We propose a set of clean, consistent and easy-to-use interfaces between the application and the direct solver libraries. Each interface takes care of dealing with the direct solver library, in a manner that is transparent to the application developers, and automatically manages matrix formats, data layout, and calling conventions. Our design is based on the following requirements:

(1) *Simplicity of usage:* Solving linear system (1) in a language like MATLAB is very easy, i.e. one just writes `x = A \ b`. It should not be much more difficult in a production code;

(2) *Flexibility:* More than one algorithm/library must be available, for both serial and parallel architectures;
(3) *Efficiency:* The overhead due to the framework must be minimal.

The basic ideas of our design are indeed quite old, and can be traced back to almost 30 years ago [Duff and Reid 1979; George and Liu 1979]. More recently, articles [George and Liu 1999; Dobrian et al. 1999] discussed the usage of abstract interfaces and OO design for the direct solution of sparse linear systems. We extend these ideas by abstracting the concepts to a higher level, and making the interfaces independent of the supported libraries. The interfaces are presented and implemented as a set of C++ classes using several well-known design patterns [Gamma et al. 1995]. C++ [Stroustrup 1986] supports object-oriented programming, and it is relatively easy to interface FORTRAN77, FORTRAN90 and C libraries with C++ code. The C++ language supports abstraction through classes, inheritance and polymorphism. For application developers, abstraction is important because it brings simplicity, by allowing components with a minimal interface. It also ensures flexibility because it decouples the different algorithmic phases from the data structures. Finally, abstraction allows extensibility in the sense that new (yet to be developed) libraries can be easily added, at almost no cost to the application developer. Another candidate language could have been FORTRAN90, but it does not have inheritance and polymorphism.

Regarding the parallel computing mode, we consider parallel architectures with distributed memory, and we suppose that the message passing interface MPI [Gropp et al. 1998] is adopted. This approach is followed by several scientific libraries, see [Heroux et al. 2005; Balay et al. 2004; Falgout and Yang 2002]. As a result, the presented design can be easily interfaced with the aforementioned projects.

The paper is organized as follows. Section 2 introduces the basic concepts of direct solution algorithms. Section 3 describes the requirements and the design of the proposed interfaces. A concrete implementation is addressed in Section 4. Section 5 reports some numerical results that quantify the overhead required by the generality of the approach. Two examples of application are reported in Section 6. Section 7 outlines the conclusions.

2. BRIEF OVERVIEW OF DIRECT SOLUTION METHODS

This section gives a broad view of the challenges of Gaussian elimination algorithms for distributed sparse linear systems. For more details on the subject, the interested reader should consult the specialized literature, for example [Duff et al. 1986; Duff 1997] and the references therein.

Following [Dongarra et al. 1998, Chapter 6], a direct solution algorithm for sparse matrices can be divided in the following phases:

(S0) Preordering: This phase determines a reordering of the original matrix such that fill is reduced in the L and U factors. This phase is difficult to parallelize and is typically computed redundantly on a parallel machine, or on a single processor with results then broadcast to other processors.
(S1) Analysis: Here the factorization is computed using the structure only, producing the patterns of L and U. In particular, many solvers use this phase to

identify dense supernodes that will improve cache and register performance; and also determine elimination trees for additional parallelism during the numerical phases (S2) and (S3).

(S2) Numerical factorization: In this phase, the actual values of L and U are computed. In most cases this phase is by far the most expensive in terms of serial operation count. However, this phase also tends to have the best single processor and parallel machine utilization of any phase. Thus, on high-performance computers, the wall-clock time of this phase will improve greatly, relative to other phases. However, it is rare to see more that a factor of 50 times speedup, even on hundreds of processors.

(S3) Forward/Back solve: This phase finds a solution x given b. Compared to the factorization, the serial cost of this a solve phase is low, perhaps by a factor of 10–100 or more. However, this phase tends to have poor parallel scaling and does not perform as well as the factorization phases on modern processors. Therefore, the cost of repeated forward/back solves (using the same factors L and U) on a parallel machine may actually become the most expensive phase.

Some codes combine phases (S1) and (S2). Phase (S0) is typical of sparse matrices: since A is sparse, L and U are still sparse, though they may have some fill-in, i.e. non-zero entries which are zero in A. Therefore, the factorization is performed on PAQ. In a typical solver for an unsymmetric matrix, the column permutation, Q is chosen to minimize fill-in, while P, the row permutation, is chosen to maintain numerical stability. Many ordering methods exist to reduce fill-in, for example multiple variations on minimum degree orderings and graph partitioning algorithms. Solvers designed for symmetric and nearly symmetric matrices typically use symmetric permutations to maintain symmetry. No single ordering method is best for all matrices, nor has a heuristic been found that consistently chooses the best ordering [Baumann et al. 2003; Amestoy et al. 2004].

Since pivoting adds complexity, which significantly increases execution time, many solvers offer options to reduce the cost of pivoting [Li and Demmel 1998; Schenk and Gärtner 2004b]. Some equilibrate the rows and columns of the matrix to improve diagonal dominance. Many codes will consider the effect on execution time when choosing a pivot, accepting some loss of numerical stability [Malard 1991]. Typically this just means accepting the diagonal pivot if it is within some threshold of the best available, but some codes will take the predicted effect on execution time into consideration when choosing an off diagonal pivot [Davis 2004].

It is possible to combine similar rows and columns into blocks to improve locality and allow high performing BLAS to be called; this combination can consider only rows and columns that are identical, or accept minor differences in the rows and columns that they treat as blocks. Such blocking can reduce the cost of symbolic factorization as well [Amestoy et al. 2001].

We have just mentioned few reasons that explained why the performance of a sparse direct solver depends on the underlying matrix, computer, application, algorithms, and libraries as well as the code and how it is compiled. Given the differences in matrix, computer, application, algorithms, and libraries, it is unlikely that a single sparse direct solver will outperform all others across all usages.

3. PROJECT DESIGN

An analysis of currently available direct solver libraries can be found in [Davis 2006]; see also [Dongarra et al. 1998, Section 6.7]. A closer look at these solvers reveals the following major differences:

(1) *Different programming languages are used:* This is true even if most projects are written in C.

(2) *Different communication paradigms are used:* Most of the reviewed libraries are serial, some are based on the MPI paradigm, others take advantage of shared memory computers.

(3) *Different matrix formats and data layouts are used:* Most of these differences are small.

(4) *Different algorithms are used:* The underlying algorithms differ and can therefore perform differently, depending on the considered matrices and computer architectures.

Despite these differences, all solvers are accessed within a given application by using a sequence of steps similar to steps (S0)–(S3) of Section 2:

(A1) Definition of the sparsity pattern of the linear system matrix;

(S0/S1) Computation of the symbolic factorization, which includes preordering and analysis. The symbolic factorization refers to all operations that can be performed by accessing the matrix structure only (without touching the values of the matrix entries);

(A2) Definition of the values of the linear system matrix;

(S2) Computation of the numeric factorization, that is, the computation of the entries of the factored terms;

(A3) Definition of the right-hand side b;

(S3) Solution of the linear system, that is, computation of x.

Steps (A1)–(A3) are application-dependent and will not be discussed here; instead, our aim is to standardized steps (S0/1), (S2) and (S3) by adding an intermediate layer between the application and the direct solver libraries. The design discussed below uses several common design patterns [Gamma et al. 1995]. The first is the *Builder Pattern*, by which we define an abstract class whose methods are steps (S0)–(S3). Because each of these steps could in principle be replaced with an alternate algorithm, our design also represents the *Strategy Pattern*. Finally, we use the *Factory Pattern* as a means of selecting a specific concrete solver instance by use of a string argument.

Presently we introduce a set of abstract classes, which will be used to define interfaces to the data layout of distributed objects, vectors, matrices, linear system, and the solver. The design is reported here as a set of C++ classes, but the concepts are more general and the discussion is largely language-independent.

The first class that we need to introduce is a `Map`, defined by

INTERFACE 3.1. *The* `Map` *class will contain the following methods:*

—`int GetNumMyElements() const` *returns the number of locally owned elements;*

—`int GetNumGlobalElements() const` *returns the global number of elements;*

—`int GetGID(const int LID) const` *returns the global ID of the local node* `ID`*.*

A `Map` defines the local-to-global numbering. We require that any global ID is assigned to exactly one processor. `Map`'s are used to define the layout of distributed vectors and matrices.

The next class is the `Vector` class, which specifies the interfaces for distributed vectors[1]. We assume that `Vector`'s locally exist as a `double` arrays. A simple set of methods for the `Vector` class is reported by

INTERFACE 3.2. *The* `Vector` *class will contain the following methods:*

—`double* GetValues()` *returns a pointer to the local array of values;*

—`const Map& GetMap() const` *returns a reference to the underlying Map object.*

As regards the linear system matrix A, we do not impose any matrix format, rather we specify an abstract interface to *query* for matrix elements. The only assumption is that the storage format adopted by the application allows a fast access to all the nonzero elements in a given (locally owned) row. This class, called `RowMatrix`, is a fairly general sparse matrix interface. For distributed matrices, we assume that each row is owned by exactly one processor; rows can be distributed in arbitrary ways among the available processors. As such, the locally owned matrix can be decomposed as

$$A_i^{(loc)} + A_i^{(ext)}, \tag{2}$$

where $A_i^{(loc)}$ represents the square submatrix of elements whose row and column correspond to locally hosted rows, while $A_i^{(ext)}$ contains matrix elements with locally owned rows and non-locally owned columns. The global elements associated with the columns contained in $A_i^{(ext)}$ are often called *ghost nodes*. The abstract matrix interface is as follows.

INTERFACE 3.3. *The abstract interface to the distributed square matrix A will contain the following methods:*

—`int GetNumMyRows() const` *returns the number of locally hosted rows;*

—`int GetNumGlobalRows() const` *returns the global number of rows;*

—`int GetNumGhostNodes() const` *returns the number of ghost nodes;*

—`void UpdateMyGhostNodes(Vector& x) const` *updates the values of ghost nodes in the input vector* `x`*;*

—`int GetNumMyRowEntries(const int ID)` *returns the number of nonzero of the (locally owned) row* `ID`*;*

—`int GetMyRow(const int ID, const int Length, int& NumEntries, int* Indices, double* Values)` *copies the column IDs and values of all nonzero elements of the locally hosted row* `ID` *in the user's allocated arrays (of length* `Length`*). The output variable* `NumEntries` *returns the number of nonzeros in the row. If* `Length < NumEntries`*, then returns -1, otherwise returns 0.*

[1]For the sake of simplicity, this paper focuses on vectors and not on multi-vectors, that is, a collection of vectors with the same `Map`. However, the presented design can be straightforwardly extended to tackle multi-vectors as well.

—`const Map& GetRowMap() const` *returns a reference to the map;*

—`bool IsSymmetric() const` *returns* `true` *if only the upper triangular part of the matrix is accessed through* `GetMyRow()`, `false` *otherwise.*

Interface 3.3 defines a standard set of methods to "view" matrix elements through row extractions. This approach is used, for instance, in [Heroux 2002; Sala et al. 2004; Duff et al. 2002; Luján et al. 2000]. We decided not to adopt iterators in `GetMyRow()` to make the code easier to use, write, and interface with "classical" C or FORTRAN. Besides, since this method will be mostly used to convert the matrix into a specific solver's format, the use of iterators would have not increased the performance.

We now specify the basic requirements for the interface to a linear problem.

INTERFACE 3.4. *The* `LinearProblem` *interface will contain the following methods:*

—`void SetMatrix(RowMatrix* A)` *sets the linear system matrix;*

—`void SetX(Vector* x)` *sets the solution vector;*

—`void SetB(Vector* b)` *sets the right-hand side vector;*

Interface 3.4 can be easily extended to standardize operations like scaling or reordering.

Finally, we can introduce the `solver` interface class, which is defined as follows:

INTERFACE 3.5. *The* `Solver` *interface will contain the following methods:*

—`void SetLinearProblem(LinearProblem* LP)` *sets the linear problem to solve;*

—`void SetParameters(List)` *specifies all the parameters for the solver;*

—`int SymbolicFactorization()` *performs the symbolic factorization, that is, all the operations that only require the matrix graph and not the actual matrix values;*

—`int NumericFactorization()` *performs the numeric factorization, that is, it computes the matrices L, D and U by accessing the matrix values. Both the solution and the right-hand side vectors are not required in this phase;*

—`int Solve() const` *solves the linear system. This phase requires the solution and the right-hand side vectors.*

—`List GetParameters()` *returns a list of output parameters, to provide feedback from the supported library.*

In the above interface, `List` is any container that can be used to specify parameters. A flowchart of the factorization processes within a typical application is reported in Figure 1.

4. A CONCRETE IMPLEMENTATION

We now describe a concrete implementation of the interfaces of Section 3, as implemented in the AMESOS[2] package [Sala 2004]. AMESOS, developed by M. Heroux, K. Stanley, M. Sala, and R. Hoekstra, is distributed within the Trilinos project [Heroux et al. 2005] and is available for public download [Sala et al. 2006].

[2]AMESOS is a Greek term that loosely translated means "direct."

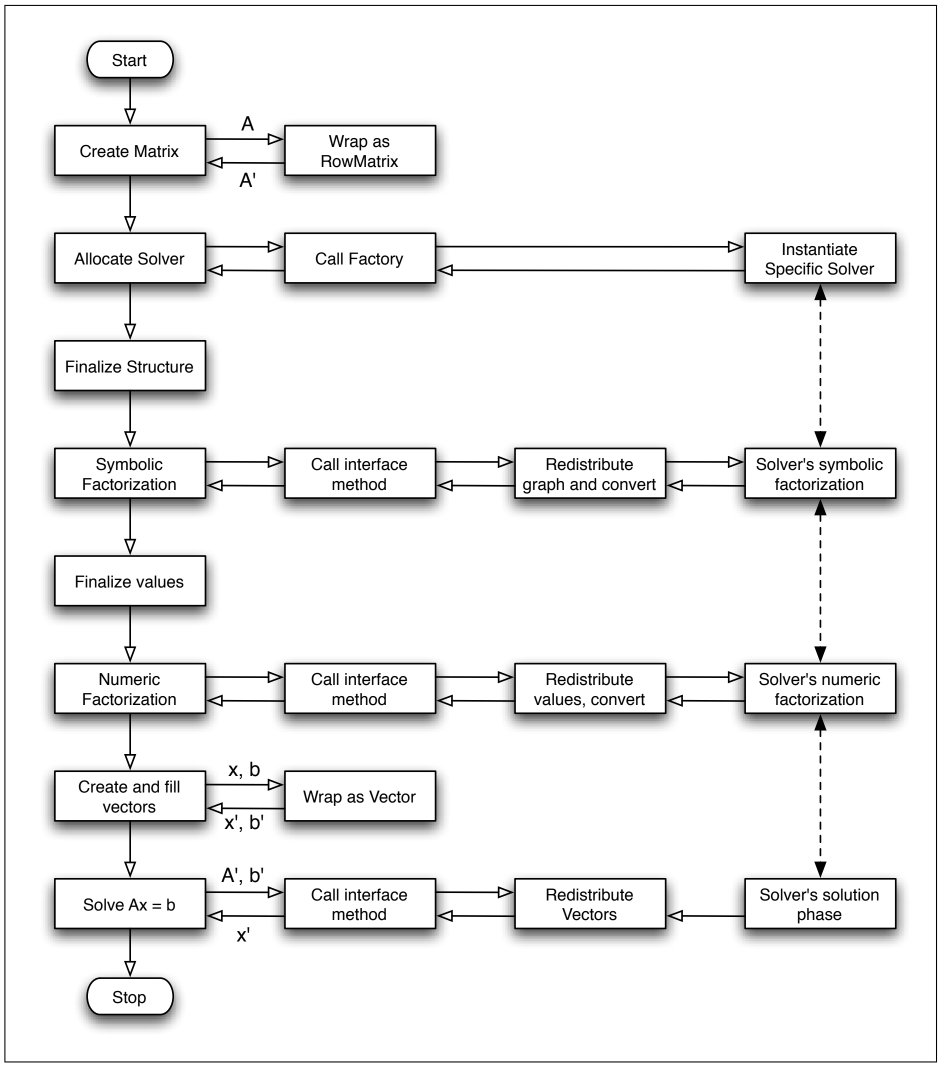

Fig. 1. Flowchart of linear system solution. A, x and b represent objects defined in the application, while A', x' and b' are the corresponding wrappers that satisfy Interfaces 3.2 and 3.3. Dashed lines mean that a given phase uses data defined in another phase. From left to right, the first column represents phases occurring in the application; the second column wrapping and calls to generic interface methods; the third column actions occurring in the solver interface, while the fourth column calls to the supported direct solver library.

We have considered the following direct solvers: KLU[3] [Davis and Palamadai 2005], UMFPACK [Davis 2003], SuperLU and SuperLU_DIST [Demmel et al. 2003] DSCPACK [Raghavan 2002], MUMPS [Amestoy et al. 2003], TAUCS [Irony et al. 2004; Rotkin and Toledo 2004; Rozin and Toledo 2004], and PARDISO [Schenk

[3]KLU source code is distributed with Amesos.

```
#include "Amesos.h"
#include "mpi.h"
#include "Epetra_MpiComm.h"
...

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  Epetra_MpiComm Comm(MPI_COMM_WORLD);

  Epetra_Map* Map = <create map here, Interface 3.1>;
  Epetra_MultiVector* x = <create solution vector here, Interface 3.2>;
  Epetra_MultiVector* b = <create right-hand side here, Interface 3.2>;
  Epetra_CrsMatrix* A = <create matrix here, Interface 3.3>;

  Epetra_LinearProblem Problem(A, x, b); // Interface 3.4

  Amesos Factory;                  // create a factory class
  string SolverType = "Umfpack";   // selected interface
  Amesos_BaseSolver* Solver;       // generic solver object, Int. 3.5
  Solver = Factory.Create(SolverType, Problem); // create solver

  Teuchos::ParameterList List;     // allocate container for params,
  List.set("PrintTiming", true);   // set one in the container, then
  Solver->SetParameters(List);     // pass the container to the solver

  Solver->SymbolicFactorization(); // symbolic factorization
  Solver->NumericFactorization();  // numeric factorization
  Solver->Solve();                 // linear system solution
  delete Solver;

  MPI_Finalize();
  return(EXIT_SUCCESS);
} // end of main()
```

Fig. 2. Example of code using AMESOS. The code uses the AMESOS interface to UMFPACK to solve the linear system. The creation of the matrix, solution and right-hand side are not reported.

and Gärtner 2004a; 2004b]. An interface to MA28 is under development. These libraries use different algorithms that are representative of a far wider range of codes. Also, these supported libraries are among the best codes publicly available, and are widely used.

AMESOS also includes interfaces to LAPACK and ScaLAPACK. The rationale is that sparse libraries are much more complicated than libraries for dense systems, and the question of how much is gained with respect to dense solvers is often asked. Dense solvers are usually more robust than sparse solvers and could be used as last-resort in extreme cases. By adding support for LAPACK and ScaLAPACK, users can experiment with these solvers, which might outperform sparse libraries in some cases (for instance, if applied to small matrices, or almost dense matrices).

To increase portability, AMESOS is configured using Autoconf [Free Software Foundation 2004a] and Automake [Free Software Foundation 2004b]; each interface

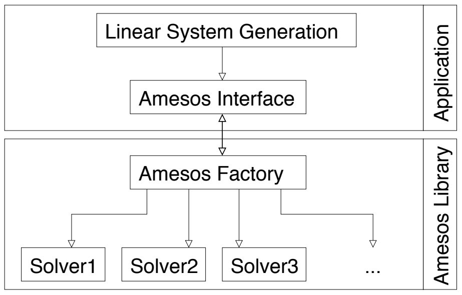

Fig. 3. Connection between a generic application and AMESOS.

can be enabled or disabled at configure time. Users can take advantage of the bug-tracking tool Bugzilla [The Mozilla Organization 2004] to provide feedback or request improvements.

AMESOS provides and abstraction for Interface 3.5, as well as all its concrete implementations (for a little more than 13,000 code lines, including comments and Doxygen documentation), and it takes advantage of the EPETRA package [Heroux 2002] to implement Interfaces 3.1, 3.2, 3.3 and 3.4. EPETRA defines several concrete implementations of Interface 3.3, making it easy to read matrices from formats like AIJ or CSR. The Standard Template Library (STL) [Wise 1996] is used to increase performance whenever possible. The AMESOS implementation of Interface 3.5 also contains methods `PrintStatus()` and `PrintTiming()` that offer a standard way to access various statistics and timings for all packages.

An example of usage of AMESOS is reported in Figure 2. Although this particular example requires MPI, AMESOS can be compiled with or without support for MPI. (Clearly, distributed solvers are available only when compiled with MPI support.) The only AMESOS include file is `Amesos.h`, which does not include the header files of the supported library. The required interface is specified by the string variable `SolverType`, and is created by using the factory class `Amesos`. Factory classes are a programming tool [Alexandrescu 2001] that implements a sort of "virtual constructor," in the sense that one can instantiate a derived (concrete) object while still working with abstract data types. The example reported in Figure 2 adopts UMFPACK as a solver; however by using the factory class, other interfaces can be created by simply changing the parameter `SolverType`. Note that the supported solver can be serial or parallel, dense or sparse: the user code still remains the same, except for the name of the solver; AMESOS will take care of data redistribution if required by the selected solver.

A generic interface between an application code and AMESOS is as represented in Figure 3: Once A, x and b of (1) are available, the application can use the factory to access any of the supported libraries.

Each AMESOS interface automatically selects the default parameters defined by the supported solver. If required, the user can tune some of the parameters by using the method `SetParameters()`. The list of supported parameters is reported in [Sala 2004].

5. NUMERICAL RESULTS

The example code of Figure 2 has shown the facility of usage of Amesos; this section, instead, aims to quantify its overhead.

Figure 4 reports the percentage of CPU time required by the Amesos interface with respect to the time required by the underlying library. We have considered the SuperLU and UMFPACK interface for the solution of all matrices in the FIDAP collection, available at [Boisvert et al. 1997]. The matrix sizes range from 27 (`FIDAP005`) to 22294 (`FIDAPM11`). All problems are solved using a 1.67 GHz G4 processor with 1024 Mbytes of RAM, running MAC OS X 10.4 and gcc 4.0.0. The table reports the percentage of the CPU time required by the interface with respect to the time required by the considered solver, and quantifies the overhead required by Amesos. We have used the default set of parameters for both solvers.

As expected, for small matrices (for example `FIDAP005` or `FIDAPM05`, of size 27 and 42, respectively) the overhead is considerable. When considering bigger matrices ($n > 3000$), then the overhead is always below 5%. All the overhead is spent in converting the matrix from the abstract format of Interface 3.3 to the format required by the library, and performing additional safety checks. Note that this overhead can indeed be reduced by adding specialized classes, that satisfies Interface 3.3 but internally store the matrix in the format required by a given solver library. Solvers derived from Interface 3.5 can perform a `dynamic_cast`, and then get the already allocated data structure containing the matrix. This solution is inelegant and requires knowledge of derived classes in the solver interface, but could greatly increase performance. Vectors are less problematic since Interface 3.2 already offers the ability to get the raw pointers required by the supported library. However, these vectors may need to be redistributed to match a given solver's requirements.

6. EXAMPLES OF APPLICATIONS

6.1 Applications to Domain Decomposition and Multilevel Preconditioners

From an abstract point of view, an algebraic domain decomposition preconditioner for the iterative solution of the linear system (1) is any preconditioner B that can be written as

$$B^{-1} = \sum_{i=1}^{M} P_i' A_i'^{-1} R_i, \tag{3}$$

where M is the number of subdomains, R_i is the restriction to subdomain i, $P_i = R_i^T$, P_i' an auxiliary prolongator operator, and A_i' an approximation to $A_i = R_i A P_i$. We refer to monographs [Quarteroni and Valli 1999; Smith et al. 1996] for a detailed overview of domain decomposition methods, and to [Saad 1996] for their algebraic interpretation.

One disadvantage of one-level domain decomposition preconditioners is that their performance deteriorates as the number of subdomains increases. Therefore, these preconditioners are completed by one or more additional (coarser) levels, as done in two-level domain decomposition preconditioners or in multilevel methods; see for instance [Brandt 1977; Hackbusch 1985].

Direct solution methods can therefore be required in two distinct phases: In the

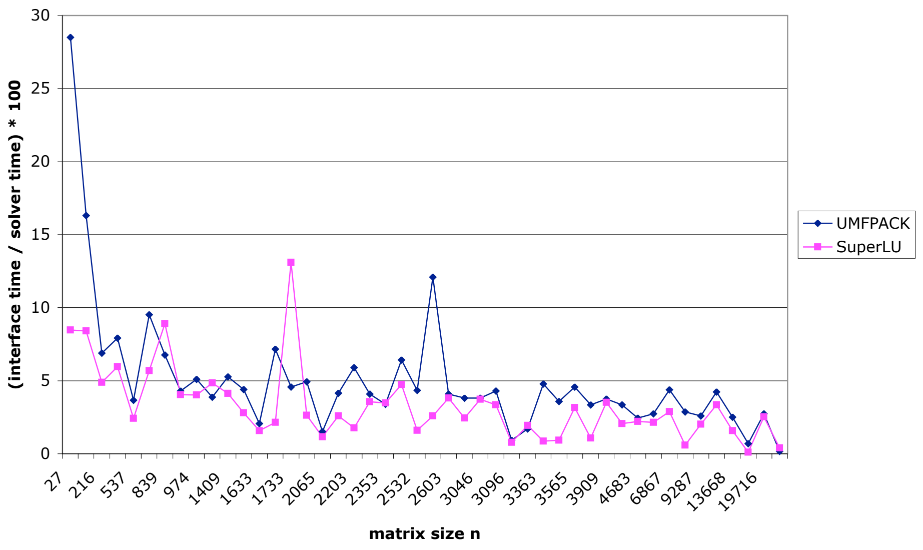

Fig. 4. Additional time required by the Amesos interface with respect to the time required by the solver, for different matrices and solvers. n represents the size of the matrix, and nnz the total number of nonzeros. The results were obtained on a G4 1.67 GHz with 1 GByte of RAM.

application of the A_i''s, and in the solution of the coarse problem. In both cases, it suffices to wrap the A_i''s or the coarse matrix to satisfy Interface 3.3 in order to access to any of the libraries supported by Amesos. This will be done by using a sequence of instructions that basically looks like the one shown in Figure 2, with the only difference that the vectors `x` and `b` must be specified just before a call to `Solve()`. The libraries IFPACK [Sala and Heroux 2005] and ML [Sala et al. 2004] are two examples of preconditioning packages adopting this approach.

6.2 Scripting Language Interface: PyAmesos

Another advantage of the presented set of interfaces is the ease with which they can be used from inside a scripting language like Python by adopting tools like SWIG [Beazley 2003]. This approach makes all the supported libraries available to Python developers at almost no cost. For more details, we refer to the PyTrilinos documents [Sala et al. 2005; Sala 2005]. An example of usage is reported in Figure 5. The example reads a matrix in Harwell/Boeing format [Duff et al. 1989], distributes it linearly across the available processors, then calls MUMPS to solve the linear system. Python's dictionaries are used to specify parameters (in this case, the level of output). Note that, since SWIG supports intra-language class inheritance, it is possible to derive Interfaces 3.2 and 3.3 in Python, and then pass it to the Python wrapper of Interface 3.5.

7. CONCLUDING REMARKS

In this paper, we have presented a model to access direct solver libraries. The model is composed of five abstract interfaces. The advantages of this model are the following:

```
#! /usr/bin/env python
from PyTrilinos import Amesos, Triutils, Epetra

Comm = Epetra.PyComm()
Map, A, x, b, Exact = Triutils.ReadHB("fidap035.rua", Comm)

Problem = Epetra.LinearProblem(A, x, b);
Factory = Amesos.Factory()
SolverType = "MUMPS"
Solver = Factory.Create(SolverType, Problem)
AmesosList = {
  "PrintTiming": True,
  "PrintStatus": True
}
Solver.SetParameters(AmesosList)
Solver.SymbolicFactorization()
Solver.NumericFactorization()
Solver.Solve()
```

Fig. 5. Complete script that solves a linear system using AMESOS/MUMPS in Python.

—The actual data storage format of the linear system matrix becomes largely unimportant. Each concrete implementation will take care, if necessary, to convert the input matrix to the required data format. This means that the application can choose *any* matrix format that can be wrapped by the abstract matrix interface.

—Issues like diagonal perturbations, dropping, reordering or fill-reducing algorithms can be easily introduced within the abstract matrix interface. For example, a dropping strategy or a modification of the diagonals simply requires a new `GetMyRow()` method, without touching the actual matrix storage. Also, reordering techniques can be implemented and tested independently of the library used to perform the factorization.

—The actual calling sequence required by each library to factor the matrix and solve the linear system is no longer exposed to the user, who only has to call methods `SymbolicFactorization()`, `NumericFactorization()` and `Solve()`.

—Interfaces can be tested more easily because they are all located within the same library and not spread out into several application codes. The framework is also quite easy to learn and use, since a basic usage requires about 20 code lines (see the example of Figure 2).

—It is easy to compare different solvers on a given set of problems. The AMESOS distribution contains a (working) template that reads a linear system from file in the popular Harwell/Boeing format [Duff et al. 1989] and solves it with all the enabled solvers. Users can easily modify this template to numerically evaluate the optimal library for *their* problems.

—The framework can serve users with different levels of expertise, from the usage of libraries as black-box tools, to a fine-tuning of each library's parameters.

—The framework can be easily extended to incorporate libraries for the resolution of over-determined and under-determined systems. Solving such systems may involve algorithms other than Gaussian eliminations; nevertheless, the interfaces will remain almost untouched.

The generality of the proposed model comes at a price. The presented model has the following limitations:

—Overhead may be introduced when converting or redistributing the matrix and/or the vectors into the library's format. For very large matrices, this can constitute a problem especially in terms of memory consumption, but is often not a first-order concern.

—Fine-tuning of solver's parameters can be difficult since Interface 3.5 has no knowledge of the underlying solver data structure. Also, we offer no "intelligent" way of setting these parameters. Projects like the GRID/TLSE [Daydé et al. 2004] can provide insight on the choice of parameters.

—There is no standard way to convert MPI communicators defined in C to MPI communicators defined in FORTRAN90. On some architectures it is difficult or even impossible to perform such a task. Some hacks may be required.

—It is almost impossible to support different releases of a given software library, because function names usually do not change from one version to the next, making it impossible for the linker to select the appropriate version of the library.

—Some libraries offer a one-solve routine without storing any data after the solution is completed; this option is not supported by the presented design, but it could be easily added.

—There are no capabilities to obtain the L, D and U factors, or the reorderings P and Q. This is because each supported package uses a different storage format and distribution. Reordering and scaling can be made library-independent by working on Interfaces 3.3 and 3.4.

—Problematic user-package communications. Because of the high-level view, the code is safer: it is more difficult to make errors or call the solver with the wrong set of parameters. Amesos classes automatically perform safety checks, and return an error code when something goes wrong. However, it is often difficult to abstract the error messages from all supported libraries and report them in a uniform fashion. Users still need to consult the library's manual to decode the error messages.

Despite these issues, we find that the presented set of interfaces brings its users the well-known benefits of reusable libraries. Thanks to their generality, these interfaces (and the corresponding codes) can be used to easily connect intricate applications with state-of-the-art linear solver libraries, in a simple and easy-to-maintain way. From the point of view of application developers, the small amount of required code makes it very convenient to adopt a project like Amesos. For linear solver libraries' developers writing one interface for their own solver can help to make it applicable and testable to a vast range of applications.

One of our goal in the design of Amesos was to reduce the intellectual effort required to use direct solver libraries. We feel that this objective has been achieved, and the performance penalty is very limited in most cases. In our opinion, the only limitation of Amesos is that it supports double precision only, while most direct solvers allows the solution in single precision and complex arithmetics. Our model can support single precision and complex arithmetics (by using templates, for example), and a templated version of Interfaces 3.1–3.5 is under development.

Acknowledgments

The authors wish to thank R. Hoekstra, T. Davis and I. Duff for fruitful discussions and suggestions about the organization of the Amesos project. We thank Oscar Chinellato for several suggestions that helped to improved this document. Part of the work has been conducted under the financial support of the Sandia National Laboratories.

REFERENCES

Alexandrescu, A. 2001. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. 1997. Plapack: parallel linear algebra package design overview. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM).* ACM Press, New York, NY, USA, 1–16.

Amestoy, P., Duff, I., L'Excellent, J.-Y., and Koster, J. 2003. *MUltifrontal Massively Parallel Solver (MUMPS Versions 4.3.1) Users' Guide.* CERFACS, Toulouse, France.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 381–388.

Amestoy, P. R., Duff, I. S., L'excellent, J.-Y., and Li, X. S. 2001. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Trans. Math. Softw. 27,* 4, 388–421.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Croz, J. D., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2004. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory.

Baumann, M., Fleischmann, P., and Mutzbauer, O. 2003. Double ordering and fill-in for the LU factorization. *SIAM J. Matrix Anal. Appl. 25,* 3, 630–641.

Beazley, D. M. 2003. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst. 19,* 5, 599–609.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Jemmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., alker, D. W., and Whaley, R. C. 1997. *ScaLAPACK Users' Guide.* SIAM Pub.

Boisvert, R. F., Pozo, R., Remington, K., Barrett, R. F., and Dongarra, J. J. 1997. Matrix market: a web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software.* Chapman & Hall, Ltd., London, UK, UK, 125–137.

Brandt, A. 1977. Multi-level Adaptive Solutions to Boundary-Value Problems. *Math. Comp. 31,* 333–390.

Davis, T. 2006. Summary of available software for sparse direct methods. http://www.cise.ufl.edu/research/sparse/codes.pdf, Univ. of Florida.

Davis, T. A. 2003. UMFPACK home page. http://www.cise.ufl.edu/research/sparse/umfpack.

Davis, T. A. 2004. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 165–195.

Davis, T. A. and Palamadai, E. 2005. KLU: a sparse LU factorization for circuit simulation matrices. Tech. rep., Technical report, Univ. of Florida, in preparation.

Daydé, M., Giraud, L., Hernandez, M., L'Excellent, J.-Y., Puglisi, C., and Pantel, M. 2004. An Overview of the GRID-TLSE Project. In *Poster Session of 6th International Meeting VECPAR'04, Valencia, Espagne*. 851–856.

Demmel, J. W., Gilbert, J. R., and Li, X. S. 2003. *SuperLU Users' Guide*. Computer Science Division, University of California, Berkely.

Dobrian, F., Kumfert, G., and Pothen, A. 1999. The design of sparse direct solvers using object-oriented techniques. Tech. rep., Institute for Computer Applications in Science and Engineering (ICASE), Hampton, Virginia, USA.

Dongarra, J. J., Duff, I. S., Sorensen, D. C., and van der Vorst, H. 1998. *Numerical Linear Algebra for High-Performance Computing*. SIAM, Philadelphia, PA.

Duff, I. S. 1997. Sparse numerical linear algebra: direct methods and preconditioning. In *The state of the art in Numerical analysis*, I. S. Duff and G. A. Watson, Eds. Oxford University press, Oxford, 27–62.

Duff, I. S., Erisman, A. M., and Reid, J. K. 1986. *Direct methods for sparse matrices*. Oxford University Press, Inc., New York, NY, USA.

Duff, I. S., Grimes, R. G., and Lewis, J. G. 1989. Sparse matrix test problems. *ACM Trans. Math. Softw. 15,* 1, 1–14.

Duff, I. S., Heroux, M. A., and Pozo, R. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS) 28,* 2, 239–267.

Duff, I. S. and Reid, J. K. 1979. Performance evaluation of codes for sparse matrix problems. In *Performance evaluation of numerical software*. North-Holland, New York, 121–135.

Falgout, R. D. and Yang, U. M. 2002. HYPRE: A library of high performance preconditioners. *Lecture Notes in Computer Science 2331*, 632–641.

Farhat, C. and Roux, F.-X. 1992. An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems. *SIAM J. Sci. Stat. Comput. 13,* 1, 379–396.

Free Software Foundation. 2004a. Autoconf Home Page. http://www.gnu.org/software/autoconf.

Free Software Foundation. 2004b. Automake Home Page. http://www.gnu.org/software/automake.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

George, A. and Liu, J. 1999. An object-oriented approach to the design of a user interface for a sparse matrix package. *SIAM Journal on Matrix Analysis and Applications 20,* 4, 953–969.

George, A. and Liu, J. W. H. 1979. The design of a user interface for a sparse matrix package. *ACM Trans. Math. Softw. 5,* 2, 139–162.

Golub, G. H. and Van Loan, C. F. 1996. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.

Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., and Snir, M. 1998. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, USA.

Gupta, A. 2001. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. Technical Report, IBM T.J. Watson Research Center.

Hackbusch, W. 1985. *Multi-grid Methods and Applications*. Springer-Verlag, Berlin.

Heroux, M. A. 2002. *Epetra Reference Manual*, 2.0 ed. http://software.sandia.gov/trilinos/packages/epetra/doxygen/latex/EpetraReferenceManual.pdf.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. 2005. An overview of the trilinos project. *ACM Trans. Math. Softw. 31,* 3, 397–423.

IRONY, D., SHKLARSKI, G., AND TOLEDO, S. 2004. Parallel and fully recursive multifrontal supernodal sparse cholesky. *Future Generation Computer Systems 20,* 3 (Apr.), 425–440.

LI, X. S. AND DEMMEL, J. W. 1998. Making sparse gaussian elimination scalable by static pivoting. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM).* IEEE Computer Society, Washington, DC, USA, 1–17.

LUJÁN, M., FREEMAN, T. L., AND GURD, J. R. 2000. Oolala: an object oriented analysis and design of numerical linear algebra. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM Press, New York, NY, USA, 229–252.

MALARD, J. 1991. Threshold pivoting for dense lu factorization on distributed memory multiprocessors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing.* ACM Press, New York, NY, USA, 600–607.

QUARTERONI, A. AND VALLI, A. 1999. *Domain Decomposition Methods for Partial Differential Equations.* Oxford University Press, Oxford.

RAGHAVAN, P. 2002. Domain-separator codes for the parallel solution of sparse linear systems. Tech. Rep. CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University.

ROTKIN, V. AND TOLEDO, S. 2004. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software 30*, 19–46.

ROZIN, E. AND TOLEDO, S. 2004. Locality of reference in sparse Cholesky methods. To appear in Electronic Transactions on Numerical Analysis.

SAAD, Y. 1996. *Iterative Methods for Sparse Linear Systems.* PWS, Boston.

SALA, M. 2004. Amesos 2.0 reference guide. Tech. Rep. SAND-4820, Sandia National Laboratories. September.

SALA, M. 2005. Distributed sparse linear algebra with PyTrilinos. Tech. Rep. SAND2005-3835, Sandia National Laboratories, Albuquerque NM, 87185. June.

SALA, M. AND HEROUX, M. 2005. Robust algebraic preconditioners with IFPACK 3.0. Tech. Rep. SAND-0662, Sandia National Laboratories. February.

SALA, M., HU, J., AND TUMINARO, R. 2004. ML 3.1 smoothed aggregation user's guide. Tech. Rep. SAND-4819, Sandia National Laboratories. September.

SALA, M., SPOTZ, W. F., AND HEROUX, M. A. 2005. PyTrilinos: High-performance distributed-memory solvers for Python. Submitted.

SALA, M., STANLEY, K. S., HEROUX, M. A., AND HOEKSTRA, R. 2006. Amesos home page. http://software.sandia.gov/trilinos/packages/amesos, Sandia National Laboratories, Albuquerque, NM, USA.

SCHENK, O. AND GÄRTNER, K. 2004a. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report, Department of Computer Science, University of Basel. Submitted.

SCHENK, O. AND GÄRTNER, K. 2004b. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems 20,* 3, 475–487.

SMITH, B., BJORSTAD, P., AND GROPP, W. 1996. *Domain Decomposition: Parallel Multilevel methods for elliptic partial differential equations.* Cambridge University Press.

STROUSTRUP, B. 1986. *The C++ Programming Language.* Addison-Wesley, Reading, MA.

THE MOZILLA ORGANIZATION. 2004. Mozilla Bugzilla Home Page. http://www.mozilla.org/projects/bugzilla.

WISE, G. B. 1996. An overview of the standard template library. *SIGPLAN Not. 31,* 4, 4–10.