**SANDIA REPORT**

SAND2005-4239
Unlimited Release
Printed July 2005

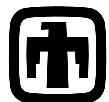
# On the Design of Interfaces to Serial and Parallel Direct Solver Libraries

Marzio Sala

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

 Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

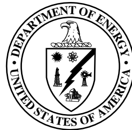
Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online



SAND2005-4239
Unlimited Release
Printed July 2005

# On the Design of Interfaces to Serial and Parallel Direct Solver Libraries

Marzio Sala
Computational Mathematics and Algorithms Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

**Abstract**

We report on the design of general, flexible, consistent and efficient interfaces to direct solver algorithms for the solution of systems of linear equations. We suppose that such algorithms are available in form of software libraries, and we introduce a framework to facilitate the usage of these libraries. This framework is composed by two components: an abstract matrix interface to access the linear system matrix elements, and an abstract solver interface that controls the solution of the linear system.

We describe a concrete implementation of the proposed framework, which allows a high-level view and usage of most of the currently available libraries that implements direct solution methods for linear systems. We comment on the advantages and limitation of the framework.

# Acknowledgments

The author would like to acknowledge the support of the ASCI and LDRD programs that funded development of Trilinos, and all the Amesos developers for their contributions to this project.

# On the Design of Interfaces to Serial and Parallel Direct Solver Libraries

## Contents

**1 Introduction** ........ **6**

**2 Review of Considered Direct Solver Libraries** ........ **7**

**3 Project Design** ........ **8**
3.1 The Abstract Matrix Interface ........ 9
3.2 The Abstract Solver Interface ........ 10

**4 A Concrete Implementation: The Amesos Project** ........ **11**
4.1 Local Solvers in Domain Decomposition Preconditioners ........ 12
4.2 Coarse Solver in Multilevel Methods ........ 12
4.3 Using Amesos from Python ........ 13

**5 Concluding Remarks** ........ **13**

# 1 Introduction

In this paper we report on the design of flexible, consistent and efficient interfaces to serial and parallel direct solver algorithms. We consider serial or distributed linear systems of type

$$AX = B, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a real, square matrix, and $X \in \mathbb{R}^n$ and $B \in \mathbb{R}^n$ are the solution and right-hand side multi-vector, respectively.

Generally speaking, a direct solver algorithm for (1) is any technique that defines three matrices, $L$, $D$ and $U$, so that $A = L\,D\,U$, and the linear systems with matrices $L$, $D$ and $U$ are easy to solve. Generally, $L$ is a lower triangular matrix, $U$ is an upper triangular matrix, and $D$ is a diagonal matrix (or possibly the identity matrix).

Developing an efficient, serial or parallel, direct solver for general matrices is a challenging task that has been a subject of research for the past four decades. Even if significant breakthroughs have been made during this time, it is still very challenging to implement direct solver algorithms even on a single processor, yet alone on multiprocessor machines. In general, the implementation of direct algorithms requires much more expertise, longer time, and more resources than other numerical analysis algorithms. In this article we do not discuss factorization algorithms, and we refer to [1, 13] and the references therein for more details. Here, our basic assumption is that one or more factorization algorithms are available in the form of software libraries, and we focus on the *usage* of these libraries. Our final goal is the definition of an easy-to-use and efficient framework between them and the applications.

Is such a framework really useful? After all, it is quite simple to write an *adaptor* between a given application code and a direct solver library: basically, one has to form the linear system matrix using the storage format required by the library, then call the correct sequence of instructions to factorize the matrix and solve the linear system. However, it is easy to see that this approach lacks of scalability: Should $n$ applications need to write adaptors to $m$ libraries, then $n \times m$ adaptors have to be developed, tested, and maintained; see Figure 1. This leads to:

- *Replicated Code.* Direct solution methods are required by several numerical algorithms. An incomplete list would include implicit time-marching schemes, Newton-like methods, multilevel and domain decomposition preconditioners. Writing an adaptor from each of these applications to a given library (or to a set of libraries) leads to replicated code, which is difficult to test, debug, and maintain.

- *Difficult Testing.* It is often difficult to choose the best direct solver library among the several existing projects. In some cases, a theoretical analysis of the problem at hand can suggest the right algorithm; alternatively, one can consider numerical comparison available in the literature on test matrices. Often, however, the best choice is to test a given solver on the application, architecture, and data set of interest, and this can be done only if an adaptor to a library implementing the given method is already available.

- *Maintenance Problems.* Including the adaptors within the application code requires the application developers to take care of several details, like matrix format, memory management, calling sequence, header files, and so on, that can vary from one version to the next.

A better solution is to introduce an intermediate level between the applications and the direct solver libraries, so that each direct solver library can be *encapsulated* within an adaptor; see Figure 2. Each adaptor will take care of interfacing each application to a given solver. The model we analyze in this paper consists of two abstract layers: an *Abstract Matrix Interface* (AMI), to which each application has

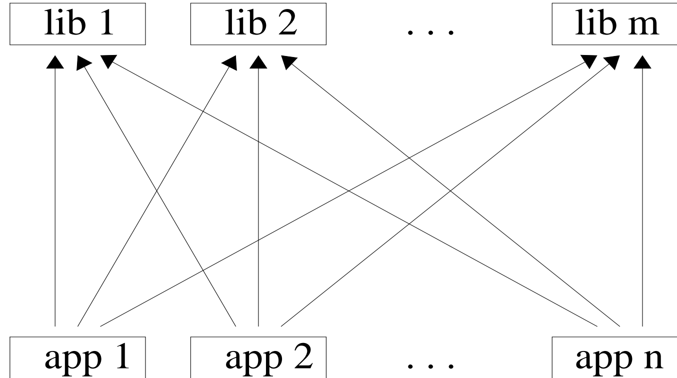



**Figure 1.** Basic software model for interfacing $n$ applications with $n$ direct solver libraries. Each arrow indicates an adaptor. $m \times n$ total adaptors are required.

to interface, and an *Abstract Solver Interface* (ASI), based on the AMI; see Figure 2. The advantages of this approach are the following. First, only $m + n$ adaptors have to be written, instead of $m \times n$. Second, the $m$ adaptors from the ASI to the direct solver libraries can be distributed within a library that does not depend on any specific application (and therefore application developers do not need to write and maintain). This enhances code usage, and allows a better testing and debugging of the interfaces. Third, each application just have to write *one* adaptor only, from its matrix format to the AMI, to automatically access all the supported adaptors. Finally, if a adaptor from the ASI to a new solver library is added, then all application can easily take advantage of this new library.

This document presents how the AMI and the ASI should be defined. Our starting point is the rich set of libraries made available by several researchers. These projects are briefly reviewed in Section 2. The interfaces are presented in Section 3. A concrete implementation of these interfaces is presented in Section 4. Note that we explicitly avoided to report any numerical comparison using these libraries, since this task is extremely challenging and application dependent. Finally, conclusions are drawn in Section 5.

# 2 Review of Considered Direct Solver Libraries

This Section gives an overview of some of the direct solver libraries we have considered to develop our set of interfaces. Our interest in mainly in sparse matrices; however, we have also considered two well-known libraries for dense matrices. For processor communication, our main interest is in MPI-based libraries.

The direct solver libraries we consider are LAPACK, a suite of serial solvers for dense matrices, written in FORTRAN77, UMFPACK [8], a C package for serial sparse matrices, PARDISO [26, 27], a package to solve large sparse matrices on shared memory multi-processors, TAUCS [17, 20, 21], a C code for symmetric matrices, SuperLU and SuperLU_DIST [9], a C code, serial or MPI-based, for generic matrices, MUMPS [2], a FORTRAN90, MPI-based code for general matrices, DSCPACK [19], a C, MPI-based code for symmetric matrices, and ScaLAPACK [5], a FORTRAN code for the parallel solution of dense linear systems.

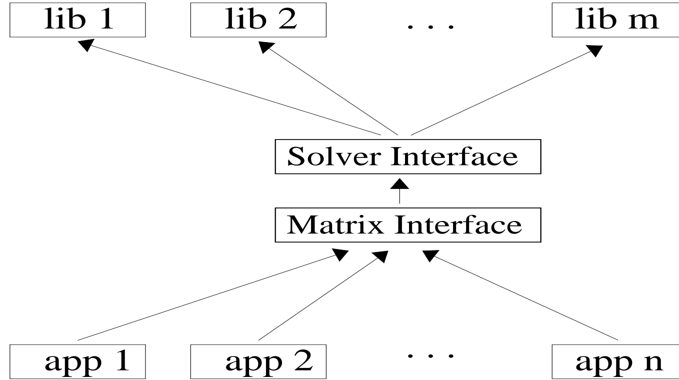



**Figure 2.** Proposed software model for interfacing $n$ applications with $m$ direct solver libraries. An intermediate level, composed by two abstract interfaces, is added between the applications and the libraries. Each arrow indicates an adaptor. $m + n$ total adaptors are required.

We have chosen these libraries for two reasons. First, these codes use different algorithms that are representative of a far wider range of codes. Second, these libraries are among the best codes publicly available, and are widely used. Other projects, like for instance SPOOLES [3] and WSMP [13] have not been considered here. However, we do not expect major problems to write adaptors for these libraries (as well as for new, future projects).

By analyzing these project, one can observe that:

- *Different programming languages are used.* The software projects reviewed in this Section are written in FORTRAN77, FORTRAN90 and C. Interfacing C with FORTRAN77 is quite easy, while interfacing C with FORTRAN90 is still quite problematic. Only one project is written in FORTRAN90, and it offers a C interface.

- *Different communication paradigm are used.* Most of the reviewed libraries are serial, some are based on the MPI paradigm, some other can take advantage of shared memory computers.

- *Different matrix formats are required.* Some libraries require the so-called CSR format, other the so-called COO format. LAPACK wants the matrix as a dense, FORTRAN77 matrix, while ScaLAPACK as a distributed dense matrix.

- *Different data layout.* Not all solvers can take advantage of parallel environments. Those that do, often require different distribution of data.

- *Different Calling Sequence.* The exact calling sequence varies widely from one library to the next. Some libraries offer more than one way to access the factorization and solution phases.

# 3 Project Design

We aim to construct a framework that allows a transpart and consistent usage of all the libraries presented in Section 2, and makes it easy to extend the support to other (new) projects. We require the following capabilities:

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

- **Flexibility:** More than one algorithm must be available, to offer optimal algorithms for small and large matrices, serial and parallel.

- **Efficiency:** The solution of (1) must be as efficient as possible, using state-of-the-art algorithms. Besides, the overhead due to the C++ design must be minimal.

In order to achieve the aforementioned goals, we introduce two generic interfaces. The Abstract Matrix Interface (AMI) is presented in Section 3.1, while the Abstract Solver Interface (ASI) is presented in Section 3.2. We do not define any Abstract Vector Interface (AVI). In principle, an AVI could favor the generality of the design; however, in our opinion this would come at a too high computational cost, and therefore we simply assume that vectors (and multi-vectors) are defined as arrays.

## 3.1 The Abstract Matrix Interface

The AMI defines how the access the matrix elements. Although converting a matrix from one given format to another is not a difficult operation, it is still a burden that can slow down or prevent the usage of a package. Our object is to define a unified interface so that the user does not have to worry about the internal matrix format of a given package. As concerns the data layout, we would like to have the be able to let users provide the linear system as it is most convenient to their needs, then let the framework will take care of data redistribution.

The most common approach to distribute the linear system matrix $A$ across the processor is to assign each row to a different processor. Therefore, each local 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. The number of columns of $A_i^{(ext)}$ containing at least one nonzero element defines the number of the so-called *ghost nodes*.

The abstract matrix interface is as follows. Function names are reported without the list of parameters to simplify the discussion.

**Interface 3.1 (Abstract Matrix Interface).** *The abstract interface to the distributed square matrix $A$ will contain the following methods:*

- `NumMyRows()` *returns the number of locally hosted rows;*

- `NumGlobalRows()` *returns the global number of rows;*

- `NumGhostNodes()` *returns the number of ghost nodes;*

- `UpdateGhostNodes()` *updates the values of ghost nodes in the input vector* `v`*;*

- `ExtractMyRow()` *copies the columns and values of all nonzero elements in locally hosted row in the user's allocated arrays.*

### 3.2 The Abstract Solver Interface

The ASI defines how to control the direct solver itself. We recall that our main goal is the make the calling sequence required to factorize and solve (1) as clean and consistent as possible. To fulfill this design requirement, we split the solution of linear system (1) into the following steps:

1. Definition of the sparsity pattern of the linear system matrix;
2. Computation of the symbolic factorization;
3. Definition of the values of the linear system matrix;
4. Computation of the numeric factorization;
5. Definition of the values of the right-hand side;
6. Solution of the linear system.

Steps 2, 4 and 6 require access to the matrix, which we suppose to adhere to Interface 3.1. Although the concrete implementations of Steps 2, 4 and 6 depend on the supported library, they can be defined as general-purpose methods. The abstract solver interface can be defined as follows.

**Interface 3.2 (Abstract Solver Interface).** *The abstract solver interface will contain the following methods:*

- `SetMatrix()` *sets the linear system matrix;*
- `SetLHS()` *sets the solution multi-vector;*
- `SetRHS()` *sets the right-hand side multi-vector;*
- `SetParameters()` *specifies all the parameters for the solver;*
- `SymbolicFactorization()` *performs the symbolic factorization, that is, all the operations that do only require the matrix graph and not the actual matrix values;*
- `NumericFactorization()` *performs the numeric factorization, that is, computes the matrices $L$, $D$ and $U$ by accessing the matrix values. Both the solution and the right-hand side are not required in this phase;*
- `Solve()` *solves the linear system. This phase requires the solution and right-hand side vectors.*

Note that the user can still toggle the parameters of a given solver by using method `SetParameters()`. Note also that, by splitting the solution phase into `SymbolicFactorization()`,`NumericFactorization()` and `Solve()`, one can reuse the symbolic factorization, or the numeric factorization, or both, for several calls to `Solve()`. If a solver does not require any symbolic factorization (for example, LAPACK), then the adaptor will not perform any operation in this method.

**Remark 1.** *The reader might wonder why we have limited our attention to direct methods only, and we did not include iterative methods as well in the definition of the abstract interfaces. The reason is that, although being often more performant in terms of CPU time and memory usage, iterative solvers are less robust and mich less black-box than direct methods. Most iterative methods are developed for PDE-like problems, and might have poor performances if applied to more general matrices. However, libraries based on the abstract matrix interface exist, as later discussed in Sections 4.1 and 4.2.*

```
#include "Amesos.h"
#include "Amesos_BaseSolver.h"

...

Epetra_LinearProblem Problem(&A, &X, &B);
Amesos_BaseSolver* Solver;
Amesos Factory;
char* SolverType = "SuperLU";
Solver = Factory.Create(SolverType, Problem);

Solver->SymbolicFactorization();
Solver->NumericFactorization();
Solver->Solve();
```

**Figure 3.** Example of usage.

# 4 A Concrete Implementation: The Amesos Project

The interfaces presented in Section 3 have been implemented in the Amesos package, developed by M. Sala, K. Stanley, M. Heroux and R. Hoekstra. We refer to the Ameso reference guide [23] and the Amesos web page for more details. Here, we briefly present the project.

Interfaces 3.1 and 3.2 have been implemented as C++ pure virtual classes. The choice of the C++ language was due to its flexibility, the easiness of interfacing with C and FORTRAN, and the availability of mature C++ compilers on almost all architectures of interest. As all other Trilinos packages, Amesos is configured and built using the GNU autoconf [11] and automake [12] tools, to make the compilation of the adaptors easy on a large variety of architectures.

Interface 3.1 is defined by the pure virtual class `Epetra_RowMatrix` of the Epetra package; see [16]. In fact, Epetra provides several concrete implementations of Interface 3.1, and several application codes already make use of Epetra classes. Besides, Epetra makes it very easy to redistribute matrices and vectors.

Interface 3.2 is defined by the pure virtual class `Amesos_BaseSolver`, and it is distributed (together with several concrete implementations) within Amesos.

A fragment of code using Amesos is as reported in Figure 3. Let us suppose that `A` is an `Epetra_RowMatrix`, and `X` and `B` are two `Epetra_MultiVector`'s. The Amesos implementation requires an `Epetra_LinearProblem`, which is a light-weight container of the linear system matrix, the solution multi-vector, and the right-hand side multi-vector. Once the linear problem has been created, we can create a generic Amesos object using the factory class, so that the `Solver` object is seen as an instance of the `Amesos_BaseSolver` by the application code. Note that only one parameter is related to the actual choice of the solver; therefore, new solvers can be added or tested simply by changing the value of `SolverType`.

The Amesos package is used by several applications. Section 4.1 and 4.2 present how the IFPACK package [24] and the ML package [25] can take advantage of Amesos. Section 4.3 show how Amesos classes can be accessed from Python.

### 4.1 Local Solvers in Domain Decomposition Preconditioners

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

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

Let $V = \mathbb{R}^n$, and let $V_i \subset V, i = 1, \ldots, M$ be a decomposition of $V$, such that $\cup_{i=1}^{M} V_i = V$, and $R_i : V \to V_i$ a restriction operator from $V$ to the subspace $i$ (of size $n_i$), and $P_i : V_i \to V, P_i' : V_i \to V$ two prolongator operators. $A_i'$ is an approximation to $A_i = R_i A P_i$.

Preconditioners of type (3) constitute an example of one-level domain decomposition preconditioners if the spaces $V_i$ are properly chosen, so that each $V_i$ contains the unknowns defined on $\Omega_i$, a contiguous subset of the computational domain $\Omega$. Parallelizing (3) is simply obtained by assigning each subdomain to a different processor. Often, parallel codes adopt a similar approach to distribute the linear system matrix, making (3) a natural candidate for the definition of the preconditioner.

The first domain decomposition algorithm has been proposed in 1870 by Schwarz to prove the solution of PDE problems on complex shape domains. In modern terms, they have been re-discovered in the late 80's, and since then widely used by pure mathematicians and computer scientists. We refer to monographs [18, 28] for a detailed overview of domain decomposition methods, to [22] for the algebraic interpretation,

Several recent projects have addressed the definition of parallel algebraic domain decomposition preconditioners. Among the most successful, we here recall the Aztec library [29], the PETSc library [4], and the PSBLAS [10]. Several other projects are devoted to define local algebraic preconditioners, such as SPARSKIT and BPKIT [7]. A common drawback of all these libraries is that it is not trivial to add new techniques to solve the local problems with matrices $A_i'$. IFPACK, instead, offers a simple framework that allows the user to specify the local solver. Since IFPACK forms the local matrices $A_i'$ by satisfying Interface 3.1, it is possible to use the Amesos project to provide access to direct solution libraries.

### 4.2 Coarse Solver in Multilevel Methods

The ML project aims to define multilevel preconditioners, in particular of algebraic type; see for instance [6], [14], or [15]. A multigrid solver tries to approximate the original PDE problem of interest on a hierarchy of grids and use 'solutions' from coarse grids to accelerate the convergence on the finest grid. A simple multilevel V cycle consisting of 'Nlevel' grids to solve (1), with $A_0 = A$ is illustrated in below.

```
/*Solve A_k u = b (k is current grid level)            */
proc multilevel(A_k, b, u, k)
    u = S_k^1(A_k, b, u);
    if ( k ≠ Nlevel − 1)
        P_k = determine_interpolant( A_k );
        r̂ = P_k^T (b − A_k u) ;
        Â_{k+1} = P_k^T A_k P_k;   v = 0;
        multilevel(Â_{k+1}, r̂, v, k + 1);
        u = u + P_k v;
        u = S_k^2(A_k, b, u);
    else
        u = A_k^{-1} b;
```

In the above method, the $S_k^1()$'s and $S_k^2()$'s are approximate solvers corresponding to $k$ steps of pre and post smoothing, respectively. Note that a possible smoother is given by domain decomposition preconditioner, and therefore IFPACK preconditioners (possibly using Amesos to provide the local solver) can be used with the multilevel scheme. Here, our interest is on the solution of the coarse problem, which is usually performed with a direct solution method. Since it is easy to wrap an ML matrix into an `Epetra_RowMatrix` (and therefore to satisfy Interface 3.1), ML can use any Amesos supported solver.

### 4.3 Using Amesos from Python

The Python package PyTrilinos offers a wrap for Amesos objects. Using PyTrilinos, all the libraries supported by Amesos are made available to Python users. An example of usage is reported below.

First, we need to create the Amesos factory,

```
>>> Factory = Amesos.Factory()
>>> Solver = Factory.Create("Superlu", Problem)
```

Then, we can perform the symbolic and numeric factorizations using methods

```
>>> Solver.SymbolicFactorization()
>>> Solver.NumericFactorization()
```

Solution is computed using

```
>>> Solver.Solve()
```

Several parameters are available to toggle the selected Amesos solver. To specify parameters, one can use Python's dictionaries:

```
>>> Factory = Amesos.Factory();
>>> Solver = Factory.Create(Type, Problem);
>>> AmesosList = {
...  "PrintTiming": ("bool", "true"),
...  "PrintStatus": ("bool", "true")
... }
...
>>> Solver.SetParameters(AmesosList);
```

# 5 Concluding Remarks

In this paper, we have presented a model to access direct solver libraries. The model is composed by two abstract interfaces, one to access the matrix elements, and the other to manage each library. The advantages of this model are:

- The actual data storage of linear system matrix becomes inessential. Each concrete implementation will take care, if necessary, to convert the input matrix into the required data format, should the row access provided by `ExtractMyRow()` method be not sufficient. This means that the application can choose the matrix format independently as long as the abstract matrix interface is implemented.

- Issues like diagonal perturbations, reordering or dropping before the factorization can easily introduces within the AMI. For example, a dropping strategy simply requires a new `ExtractMyRow()` method — without touching the actual matrix storage.
- The actual calling sequence required by each library to factorize the matrix and solve the linear system becomes inessential. Instead, the user only has to call methods `Initialize()`, `Compute()` and `Solve()`.
- Adaptors can be tested more easily because they are all located within the same library and not spread out into several application codes.

Clearly, the model is not suitable for applications. In our opinion, the most important limitations are:

- Each adaptor automatically selects the default parameters defined by the supported solver. In most cases, these values are a robust and reliable choice for most applications. If required, the user can tune some of the parameters by using method `SetParameters()`. However, some fine-tuning can be difficult since the ASI has no knowledge of the underlying solver data structure.
- 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.
- It can be difficult to support single precision and complex arithmetics. At present, only double precision is supported by the Amesos project. Although the model can in principle support single precision and complex arithmetics (by using templates, for example), no code that implements Interface 3.2 has been written yet. A templated version of Interface 3.1 is under development within the Tpetra package.
- It is almost impossible to support different versions of a given software library, because function names usually do not vary from one version to the next, making it impossible for the linker to select the appropriate version of the library.
- Not all adaptors can be compiled and linked at the same time. Often developers of direct solver libraries take advantage of other, smaller libraries, that offer common functionalities. Typically, this happens with reordering algorithms. Unfortunately, it is not uncommon for different solver libraries to request different versions of a given reordering algorithm, all codes using the same function names. As a result, not all the adaptors can be compiled and used at the same time.

# References

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

[2] P.R. Amestoy, I.S. Duff, J.-Y. L'Excellent, and J. Koster. *MUltifrontal Massively Parallel Solver (MUMPS Versions 4.3.1) Users' Guide*, 2003.

[3] C. Ashcraft and R. Grimes. Spooles: an object-oriented sparse matrix library. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[4] S. Balay, G. Gropp, L. C. McInnes, and B. Smith. PETSc user's manual. Technical Report ANL-95/11, Argonne National Laboratory, November 1995.

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

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

[7] Edmond Chow and Michael A. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24(2):159–183, 1998.

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

[9] J. W. Demmel, J. R. Gilbert, and X. S. Li. *SuperLU Users' Guide*, 2003.

[10] S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26:527–550, 2000.

[11] Free Software Foundation. Autoconf Home Page. http://www.gnu.org/software/autoconf.

[12] Free Software Foundation. Automake Home Page. http://www.gnu.org/software/automake.

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

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

[15] W. Hackbusch. *Iterative Solution of Large Sparse Linear Systems of Equations*. Springer-Verlag, Berlin, 1994.

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

[17] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. *Future Generation Computer Systems*, 20(3):425–440, April 2004.

[18] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, Oxford, 1999.

[19] P. Raghavan. Domain-separator codes for the parallel solution of sparse linear systems. Technical Report CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University, 2002.

[20] V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30:19–46, 2004.

[21] E. Rozin and S. Toledo. Locality of reference in sparse Cholesky methods. To appear in Electronic Transactions on Numerical Analysis, August 2004.

[22] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, 1996.

[23] M. Sala. Amesos 2.0 reference guide. Technical Report SAND-4820, Sandia National Laboratories, September 2004.

[24] M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, February 2005.

[25] M. Sala, J. Hu, and R. Tuminaro. ML 3.1 smoothed aggregation user's guide. Technical Report SAND-4819, Sandia National Laboratories, September 2004.

[26] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report, Department of Computer Science, University ofBasel, 2004. Submitted.

[27] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

[28] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel methods for elliptic partial differential equations*. Cambridge University Press, 1996.

[29] R. Tuminaro, M. Heroux, S. Hutchinson, and J. Shadid. Official Aztec user's guide: Version 2.1. Technical Report Sand99-8801J, Sandia National Laboratories, Albuquerque NM, 87185, Nov 1999.