CCA-LISI: On Designing A CCA Parallel Sparse Linear Solver Interface

Fang (Cherry) Liu and Randall Bramley
Indiana University
Bloomington, Indiana
{fangliu,bramley}@indiana.edu

Abstract

Sparse linear solvers account for much of the execution time in many high-performance computing (HPC) applications, and not every solver works on all problems. Hence choosing a suitable solver is crucial step for an efficient application. Unfortunately, the best solver cannot be determined during the application development. Experiments on finding best suitable solver require a plug and play mechanism.

This work is part of the Common Component Architecture (CCA) [21] effort on designing a common interface among various parallel high performance linear solver libraries, hence componenizing them and enabling dynamical switching. The implementation of this interface provides a CCA toolkit and is reusable under CCA-compliant framework such as Ccaffeine[15].

1 Introduction

Linear system solvers are ubiquitous in scientific applications, and both iterative and direct methods play important roles in solving the large-scale systems of linear equations that arise in applications. The linear system is typically written as $Ax = b$ where A is the coefficient matrix, b is the given right hand side vector, and x is the unknown vector to be solved. Although present in all linear solvers, the *matrix* may be stored in a variety of *data structures* and rarely is stored as a single 2D array. While the linear solver packages provided by several national laboratories and universities are used widely in the scientific computing community, no single package is optimal or even works in all cases. E.g., a nonlinear PDE solver may generate a sequence of linear systems which may have widely varying characteristics and require different solution methods. A mechanism facilitating switching the solvers is needed for scientific application code which often has deeply embedded linear solvers. This is particularly true for HPC distributed memory parallel codes, which require clearly identifying the distribution of the linear system's data across processors. Minimizing the required application code changes can alleviate the difficulty of changing solvers as the problems change. Similar to the way Message Passing Interface (MPI) [24] provides a portable interface to multiple underlying communication libraries, a common linear solver interface spanning multiple solver packages can be developed to meet these requirements.

A linear solver interface is also needed for the increasingly important area of *model coupling*. E.g., fusion energy simulations are now integrating codes that model different physics of a fusion reactor [28]. The constituent codes typically assume some mode decoupling to get models that are computationally feasible. Coupling the codes may involve all of the modes from each constituent code, and may also introduce intermediate modes that do not appear in any of the physics sub-models[18]. A linear solver interface can help in creating solvers for the integrated fusion simulation that combines separate codes, and enable the usage of each code's native and specialized solvers for the subproblems corresponding to each physics sub-model.

This paper identifies the design issues in creating a single software interface LInear Solver Interface (LISI) spanning a broad range of HPC linear solvers. A linear solver component using a draft LISI specification for the interface has been written, providing multiple implementations including software from Terascale Optimal PDE Solvers — a SciDAC ISIC (TOPS), Trilinos, and SuperLU. Because interfaces are defined using Scientific Interface Definition Language (SIDL) [30], the Babel compiler from Lawrence Livermore National Laboratory (LLNL) provides language interoperability for the interfaces. In particular Babel creates bindings for Fortran 77, Fortran 95, C and C++, Java, and Python, the languages most commonly used in high-performance computing. Some terminology first: developers sometimes refer to the primary iteration as the *solver* and the *preconditioner* is a transform to make the linear system easier to solve. Others focus on the preconditioner (which

1-4244-0910-1/07/$20.00

typically takes the most time in a solve), and call the iterative phase *iterative refinement*. In this paper the first terminology is used.

2 Problem Statement

Although a large number of sparse linear solver packages are available [22], none can guarantee solving the full range of linear systems encountered in applications. Some packages focus on particular class of problems and provide efficient solvers for them. High Performance Preconditioners (HYPRE) [6] has preconditioners and solvers emphasizing multi-level methods, featuring parallel multigrid methods for both structured and unstructured grid problems. Portable, Extensible Toolkit for Scientific Computation (PETSc) [17] and Trilinos [26] are general purpose packages, but with different sets of iterative solver methods and preconditioners. SuperLU [33, 32] is a direct solver for LU factorization using sparse Gaussian elimination and is good for non-symmetric nonsingular linear system. MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [10] is a parallel sparse direct solver using a multifrontal method. Iterative and direct solvers each have advantages and disadvantages. Iterative methods may converge too slowly or not at all, and the time required to solve a linear system is not always possible to determine analytically, particularly for non-symmetric and highly non-normal [31] matrices. Direct methods are more predictable but may require excessive memory to hold a full factorization. Although commonly used for robustness, a direct solver may fail in some cases where an iterative solver succeeds. E.g., a Krylov subspace solver can handle a rank-deficient coefficient matrix in a consistent linear system, provided the initial residual lacks any components in the null space of the coefficient matrix.

2.1 Switching solvers

Historically most HPC applications were written from the ground up for a single purpose with little reuse of existing codes. When built using HPC solver packages, applications can be so tightly coupled with them that it is hard to change to another underlying solver library. But application requirements may change with time, adding new physics to underlying models or being applied to new sets of problems. Being able to change out easily the solver implementation used is important. Because application codes are constantly evolving and changing, minimizing the number of lines of code that must be changed to swap solvers is critical. As an example, in fusion energy simulations the extended magnetohydrodynamics code M3D [8] uses PETSc's sparse linear solver KSP and has at least 767 lines of code in 67 subroutines directly making calls to PETSc KSP solver. The radio frequency heating code AORSA [2] has 41 lines of code in 5 subroutines that interface with its underlying solver (ScaLAPACK [20]). A single shared linear solver interface would in principle allow exchanging the underlying solvers with no changes in the application code. Some CCA frameworks like Ccaffeine [15] even allow dynamic run-time swapping of components. Switching to a different solver library also has a learning curve, particularly when the solvers are heavily parameterized and the ways of specifying those parameters vary widely. At least three groups of users can use the proposed LISI. Numerical linear algebraists need to have low level parameter access while application users typically do not want to deal with parameter settings. Computer scientists often bridge applications and solvers so that they can work together.

Changing an application code to use the proposed LISI would also require extensive application code changes and handling a new learning curve, but the intent is that the change would only need to be made once, and afterwards substituting new solvers do not require modifying the base code.

2.2 Requirements

For usage in modern HPC applications, an interface must satisfy some minimal requirements. Those include:

Parallelism. Distributed-memory parallelism is still the only architecturally scalable system, and is the most commonly used approach. This requires partitioning a global data object (the coefficient matrix A and right hand side vector b) across multiple memory spaces. Many current HPC linear solver packages use a block row partitioning on the linear system among multiple processors, corresponding to a domain decomposition of the unknowns. Some partitioning concepts have become standard for dense (fully populated) arrays, such as cyclic, block cyclic, and arbitrary rectangular multi-dimensional block decomposition [23]. Unfortunately no standard semantics exist yet for expressing the partitioning of sparse linear systems, which is part of the reason for the complexity of exchanging solver libraries for an application.

Multilevel method support. Multilevel [37] (including multigrid and some multipole) methods are the only widely available and applicable solvers that have proved scalable in practice. Multilevel solvers such as HYPRE typically involve alternating between grids of different refinement, and this entails supporting re-entrancy at least in any implementation of a common LISI. Multilevel methods typically use different solvers on different levels and it may be more efficient to use different solvers for sub-domains solved in parallel on the same level.

User-supplied linear operators. Increasingly often scientific applications do not explicitly create a data structure for the linear system's coefficient matrix and right-hand side vector. Iterative methods based on Krylov subspaces typically only require the results of matrix-vector products, and in the case of preconditioning, the result of solving a related linear system. This opens the opportunity for "matrix-free" solvers where the application is required to perform matrix-vector products and preconditioner solves, either via callbacks or by providing a function pointer to the linear solver package. For nonlinear systems, Newton-Krylov solvers [29] use a similar approach and have been proved robust, accurate, and scalable. LISI needs to allow the application itself to provide the matrix vector products, and/or the preconditioner operations.

Language interoperability. Most scientific HPC simulations are written in Fortran, but existing solver packages are developed in C, C++ or Fortran. Several systems like SWIG [19], Matlab's MEX [14], CHASM [3], JNI [7], and f2py [4] bridge between a specified target language and other languages. The relatively new Fortran 2003 standard [5] has a C interoperability specification, although it has significant limitations on the types used and does not handle the large existing base of Fortran codes in use. Microsoft's .NET, and particularly the the Common Language Infrastructure (CLI) and The Common Language Runtime (CLR) [9] in principle also address language interoperability by first translating code to a common intermediate representation. Language interoperability also is provided by Babel, which generates stubs and skeletons for different languages based on interfaces definition in SIDL. Babel supports C, C++, Fortran77, Fortran90, Python and Java being combined in a single application.

3 Related Work

Several efforts (e.g., [1] [12] [27]) have designed shared abstract interfaces for linear solver packages, but none have yet achieved wide acceptance from the HPC community.

Equation Solver Interface. The Equation Solver Interface (ESI) [1] effort started before SIDL was designed and its interfaces were written in C++. ESI development stopped around year 2001 at least in part because of a lack of community support, but it has provided a start point for other interface design efforts. ESI specified interfaces for Matrix, Vector, Parameter, Operator, Preconditioner and IndexSpace. It was a top-down effort which started from a general mathematical expression of linear systems, rather than beginning with a particular existing package and trying to generalize it. A major reason ESI was not successful was because it was premature: no large set of applications had a driving need for it, there was no practical language-independent way of expressing the interfaces, and there was not enough sustained community involvement.

The TOPS CCA interface. TOPS [11] researchers proposed a TOPS CCA interface [12] and a TOPS Solver Component at SC05. It provides direct access to virtually all of the TOPS linear and nonlinear algebraic solvers including geometric and algebraic multi-grid. The TOPS interface separate the interface into two parts: TOPS.System and TOPS.Solver. Solver users need to implement TOPS.System interface which defines the algebraic problem. TOPS.Solver provides the solver capability such as initializing the solver by passing in command line arguments, setting up the parameters, and then solving the linear system from TOPS.System. TOPS effort is a bottom up approach, it reflects the requirements of TOPS solvers in detail, and some methods have implicit assumption on how underneath data is distributed. This interface is not general enough for most of parallel solvers which are currently widely used.

Ames CCA interface. Ames laboratory works on a CCA interface for Sparskit[27]. SPARSKIT is a serial toolkit for sparse matrix computations and is written in FORTRAN 77 and has a cumbersome interface. Componentizing it enables its wider usage in modern applications and facilitates further SPARSKIT enhancements. This effort also uses bottom-up design fashion, and the interface is mainly for SPARSKIT, and since SPARSKIT is a serial solver, the interface doesn't address parallelism. It cannot be directly used for HPC scalable solver packages. The interface is also tightly coupled with the underlying library, and it cannot be used for other libraries as a general interface.

4 CCA/SIDL

The CCA was started in 1997 as an effort to bring the component programming model to scientific users, and the CCA is a specification of the component programming pattern and the interface the components see to the underlying support substrate, or framework [16]. In the CCA, a component is defined as a collection of ports, where each port represents a set of functions that are publicly available. A port is described using SIDL [30] which is a programming-language-neutral interface definition language. There are two types of ports: *provides* ports and *uses* ports. The ports implemented by a component are called *provides* ports and other components may connect to use. The ports that a component may connect to and use are named as *uses* ports. CCA uses this *provides-uses* design pattern [25] to define interactions between the components.

CCA provides a framework which facilities:

- Reusable components
- Components assembling
- Language interoperability
- Dynamical switching components with the same interface and different implementation.

The agreement on the ports functionality must be made between multiple software development teams to facilitate seamless component composition. Designing the CCA common interface is the crucial part on component reusability, since one component is considered as reusable only if it implements some publicly available interface. The LISI defines a minimal common set of functionalities among sparse linear solver packages and is a basis for further discussion from interested parties. Successful CCA common interfaces have been designed in other areas such as CCA distributed array interface [23], TAU performance interface [35], and TSTT mesh interface [13].

Babel [36] addresses language interoperability issues to enable software developed in different languages to communicate using Interface Definition Language (IDL) techniques. An IDL describes the calling interface (but not the implementation) of a particular software library. The Babel team has developed and maintains SIDL [30] which addresses unique needs of parallel scientific computing by supporting complex numbers, dynamic multi-dimensional arrays, and the parallel communication directives required for parallel distributed components. Babel uses this interface description to generate "glue code" that allows a software library implemented in one supported language to be called from any other supported language. Currently, Babel supports Fortran 77, Fortran 90, C, C++, Python, and Java (uses ports only).

CCA component allows to be reused and assembled to the application, but it is forced to specify uses ports and provides ports in the implementation in which those information must be hard-coded.

5 Design issues and requirements

Design of a minimal common set of interfaces is not trivial, especially when it spans multiple packages. A flexible interface will have to address multiple systems' problems and requirements.

5.1 Interface Complexity

The interface tries to capture the interactions between HPC applications and solvers. The goal is to hide the underlying implementation as much as possible while preserving functionality and allowing the user flexibility. Extracting commonalities among solver packages is useful to find the common interface. For freely available sparse linear solvers, three phases are commonly used:

a) Setup of linear system data structures

b) Setup of interlinked options/choices/algorithms/parameters

c) Solve

Setting up the sparse matrix, right hand side vector, solution vector through explicitly pass-in arrays in which the interface is easy to define. However, auxiliary data structures such as preconditioners, elimination trees for direct solvers, and multifrontal stacks vary among the packages. They are hard to define a single interface for. Details that need to be considered include parameters such as preconditioner method, fill level, drop tolerances, stop tests, and restarts in Step 2. The solve itself is relatively easy to define the interface for, but a common interface on post-solve phase needs careful consideration such as how the statistics information gets returned and in what order.

5.2 Usage Complexity

The way the linear solver is used by HPC applications varies based on application requirements. There are at least five different use cases:

a) One time solve: the solver is only called once with one linear system and one right hand side (RHS) vector, the solution vector is returned when the solve phase is finished. In this case the linear system and RHS vector need not be stored for reuse.

b) Precompute reused objects such as LU factorization and symbolic factorization for sparse direct solver and ILU factors for preconditioned iterative solve. Partially reusable objects need to be stored for reuse.

c) Multiple solves with the same A and multiple RHS vectors. Both A and preconditioner are reused and RHSs are usually presented one after another. The interface needs to specify how the multiple RHSs are passed in, e.g. through a multi-dimensional SIDL array or set up one by one after each solve.

d) Multiple solves with the different coefficient matrices A. Although A differs on each solve, the A typically has the same sparsity pattern as the first solve, and the preconditioner may be still reusable. Computing the preconditioner is often the most expensive part of a linear solve.

e) Recursive calls to the solver. This case is mainly for multi-level solvers. Recursion must be addressed in the interface. There are two ways to identify the LISI's role in this context: one is that LISI will be treated as the interface to single solver and a multi-level solver developer can use LISI on each level solve. The other is where the LISI handles the multi-level solver by itself. In this case an identifier may need to be provided by the application for each level of the grid.

5.3 Input Data Structure Complexity

Unlike the dense matrix, sparse matrices are often stored in some compact ways to reduce the storage requirement. The well-known formats are: coordinate format (COO), compressed sparse row format (CSR), compressed sparse column format (CSC), modified sparse column format (MSC), modified sparse row format (MSR), etc. Some existing tools [38] provide the conversion between different sparse data formats, however none of the sparse linear solver packages provides the support for all formats.

5.4 Parallelism

Specifying how data is divided across distributed memories may need to be addressed in the interface. The Distributed Array Descriptor (DAD) [23] effort within the CCA is to design a common interface for specifying this, but it currently only addresses dense arrays. Until a DAD is created for sparse linear systems, the current proposed LISI assumes that block row partitioning is used. With some initial reordering of the rows, this is the most commonly used approach.

5.5 Matrix-free Interface

Many HPC applications use a linear solver by passing in the linear system explicitly in the form of arrays, but increasingly some high end HPC applications do not form a linear system in array format explicitly. Especially for adaptive griding and extremely large problems with limited memory this can be more efficient, but the application user is then responsible for computing matrix-vector products and preconditioner application. Both Trilinos and PETSc provide mechanisms to support matrix-free methods. Trilinos's Epetra_RowMatrix virtual class allows the application developer to implement and create their own matrix data type with a matrix vector product method. The newly created matrix object can then be passed to AztecOO solver to get the solution. PETSc allows application developer to create a new Matrix type with user-provided matrix-vector product routine, and associates the Matrix and this routine through MatShellSetOperation. LISI needs to provide this functionality.

5.6 Uses-Provides

As we discuss in Section 4, CCA deploys a *uses and provides* design pattern which requires functionality separation among the HPC solvers and applications. Since LISI describes the interaction between these two parties, how to choose the *uses* ports and *provides* ports is needed to carefully think about. Figure 1 shows three cases.

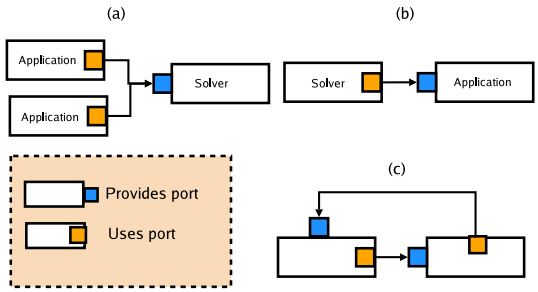

Figure 1. Possible choices for a uses-provides design pattern for sparse linear solvers

(a) Application has uses ports, solver has provides ports. The advantage is that the same solver instance can be used repeatedly and multiple applications can use the solver simultaneously. The disadvantage is that linear system must be explicitly passed into the solver. Ames CCA interface chooses this way to design their Sparskit CCA wrapper.

(b) Application has provides ports, solver has uses ports. The advantage is that matrix free method can be supported in which solver can use the *matrix-vec* product from application side to solve the linear system in a matrix free manner. The disadvantage is that application developer may not want to write all of the provides ports required since it makes them responsible for some burdensome CCA coding.

(c) In hybrid use pattern. Both application and solver have uses and provides ports. The provides port in application can provide some function such as *matrix-vec* product. TOPS interface use this hybrid provides-uses pattern, but in a way application provides the setup of linear system and solver provides the solve functionality. In their design, application always has to implement some services even matrix-free method is not required. Our design chooses the application provides port that only has matrix-free function and the rest of functions are provided by the solver for easy usage of solver.

5.7 Minimal Changes

To minimize the overhead of using LISI with current applications, interface itself must be as small as possible and not require extensive changes in the application code. However, some applications are tightly coupled with a particular library and would require more extensive restructuring.

6 Design Decision

Supporting the interface functionality requirements requires some technical issues to be resolved.

6.1 Multiple or single interface?

In CCA, the interface is exposed to other codes via a Port. LISI defines only one interface (*SparseSolver*) to be publicly accessible, but there are two ways to define how the data are passed to *SparseSolver* interface:

- Matrix data is the language build-in primitive type. In this way other components are easy to prepare the call once they have the linear system in their hands as multi-arrays. Most of sparse linear solver package written in Fortran accepts multi-array as the input parameters.
- Matrix data is object which is described as the separated interfaces such as Matrix and Vector interface. In this way, the solver component will have a good data encapsulation through object composition, but it adds a burden on using the interface, since they have to have two steps instead one to call the interface: form the pass-in object and pass the object to interface.

We decide to choose the first one since *SparseSolver* is a relatively simple interface and the data complexity is low, no need to introduce another complexity through object composition.

6.2 R-array or SIDL array?

Both r-arrays ("raw arrays") and normal SIDL arrays are supported by Babel, but the use of r-arrays is more restricted: it only has the *in* and *inout* parameter modes; for multi-dimensional array, only column-major order is supported; NULL is not allowable; the lower index is always 0; it can be used for arrays of SIDL int, long, float, double, fcomplex, dcomplex types.

From our interface requirement, we do not need *out* parameter, and we will not use multi-dimensional array since our interface expects the assembled linear system in the form of three one-dimensional arrays. Indexing starting at 0 can be treated at the component implementation by shifting by one, and R-array supported data types are sufficient for the real world application. Then r-array limitations may not hinder the LISI design, and compared with normal SIDL arrays, r-array does have advantages:

- More traditional access in each supported language.
- Developers need less or no code to translate between their array data structure and r-array data structure.
- SIDL generated APIs can have signatures similar to legacy APIs
- Less performance overhead because r-arrays can avoid calls to *malloc* and *free*.

Most importantly using the r-array instead of SIDL array as our parameter can reduce the component developer's learning overhead for SIDL array.

6.3 Single or multiple methods for parameters

A single port can have many methods defined, so the design issue is whether or not to provide separate *get* and *set* methods for each parameter. To avoid repeated parameter passing, LISI uses separate methods to set them, such as number of local rows, number of local nonzeros, and starting local row's number in the global numbering scheme. The methods are *setStartRow*, *setLocalRows*, *setLocalNNZ* and *setGlobalCol*, so that methods such as *setupMatrix*, *setupRHS* and *doSolve* need not provide those parameters on each call. This also avoid the parameter setting conflict by mistake.

6.4 Simple invocation pattern

LISI puts *uses* ports on the application side, and *provides* ports on the solver side. This seems to be a more natural approach to application users who look at solvers as utilities provided to their main code. This also makes the application side easier and less embedded into CCA implementations by simply invoking the interface to pass the linear system and RHS and get the result back. As Section 5.6 explained, this choice is somewhat arbitrary and there are arguments for reversing the uses/provides roles chosen by LISI.

6.5 Generic or specific parameter-setting methods?

Sparse direct solvers tend to have a shared terminology, e.g. Markovitz pivoting parameter, drop tolerance, etc. Similarly for sparse iterative methods, there are levels of fill in the preconditioner, stopping test, stopping tolerance,

maximum allowed iterations, etc. So an interface can be defined with specifically name methods for those. However, solvers vary in the amount of parameter control available to the application, and may have unique or unusual parameters to be set. Even when terminology matches different interpretations are taken by different solver libraries. For instance "stopping tolerance" can mean an absolute residual norm test or a residual measure normalized by the initial residual. LISI handles this by making the parameter setup methods as generic as possible, instead of giving a fixed method name for each parameter such as *setSolverMethod* and *setPreconditioner*.

7 Proposed LISI

The SIDL specification in code listing 7.2 implements the design decisions made in Section 6 and attempts to satisfy most of the requirements described in Section 5

The interface itself does not provide an implementation, but does need to mediate between an application code and solver libraries. Since the interface is written in SIDL, implementations can be written with different languages and even with different languages for a paired uses/provides port. LISI does not provide solvers itself and is just an interface and adapter implementation to them.

7.1 Design Architecture

Figure 2 shows *LISI* interface's role in the HPC application, it sits between native sparse solver packages and application codes, along with Babel generated client stub for muti-language support. The arrows indicate the calling data flow. By introducing *LISI* layer, the goal of decoupling is achieved.

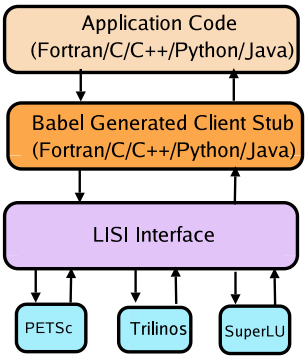

Figure 2. Interface Design architecture

7.2 Description

This package is an interface for both iterative and direct solvers. To avoid name conflict, the interface is put in package *lisi*. The *SparseSolver* interface extends from *gov.cca.Port* and it is implemented by solver components to provide the solver functionality. Enum type *SparseStruct* allows the input the array format to be chosen among CSR, COO, MSR and VBR, etc. *setStartRow*, *setLocalRows*, *setLocalNNZ* and *setGlobalCols* set the data distribution variable which are needed for the other methods. The *setupMatrix* is overloaded by the different input data format and array offset. In the implementation, it works as an adapter to convert the input data format to the libraries' internal data structure and frees up users from doing it by their own. *setRHS* sets up the RHS vector. The rest set of *setXXX* methods provide the generic way to setup the internal solver parameter, and *key* is the parameter name, the agreement on the key's name should be associated with the LISI, currently *key* is the list of *solver*, *preconditioner* *tol*, *maxits*, etc. The *solve* invokes the internal solver and returns the solution vector in parallel. As for matrix-free functionality, LISI has a *MatrixFree* interface which provides matrix-vector product method. The Enum type *ID* is used here to identify if the method is called for solver or preconditioner. Since *MatrixFree* interface will be implemented by application side, we make the assumption that data distribution information is already known.

CCA LISI SIDL Interface

```
package lisi version 0.1
{
 enum SparseStruct
 { CSR, COO, MSR, VBR, FEM,}
 enum ID
 { MATRIX, PRECONDITIONER,}

 interface MatrixFree extends gov.cca.Port{
  int matMult(in ID id,
        in rarray<double,1> x(length),
        inout rarray<double,1> y(length),
        in int length);
  }

 interface SparseSolver extends gov.cca.Port{
  int initialize(in long comm);
  int setBlockSize(in int bs);

  /* Block row partitioning */

  int setStartRow(in int startrow);
  int setLocalRows(in int rows);
  int setLocalNNZ(in int nnz);
  int setGlobalCols(in int cols);

  int setupMatrix[few_args](
    in rarray<double,1> Values(NNZ),
    in rarray<int,1> Rows(NNZ),
    in rarray<int,1> Columns(NNZ),
    in int NNZ);

  int setupMatrix[media_args](
    in rarray<double,1> Values(NNZ),
```

```
    in rarray<int,1> Rows(RowsLength),
    in rarray<int,1> Columns(NNZ),
    in SparseStruct DataStruct,
    in int RowsLength, in int NNZ);

  int setupMatrix[large_args](
    in rarray<double,1> Values(NNZ),
    in rarray<int,1> Rows(RowsLength),
    in rarray<int,1> Columns(NNZ),
    in SparseStruct DataStruct,
    in int RowsLength,
    in int NNZ, in int Offset);

  int setupRHS(
     in rarray<double,1>
        RightHandSide(NumLocalRow),
     in int NumLocalRow, in int nRhs);

  int solve(
    inout rarray<double,1> Solution(NumLocalRow),
    inout rarray<double,1> Status(StatusLength),
    in int NumLocalRow,in int StatusLength);

  int set(in string key, in string value);
  int setInt(in string key, in int value);
  int setBool(in string key, in bool value);
  int setDouble(in string key, in double value);

  string get_all();
  }
}
```

The proposed LISI leaves some open questions on the interface design, and as discussed in Section 5, some issues aren't addressed. The current LISI is the core part and provides a basis for discussions in the HPC solver community, rather than a final specification.

8 Implementation and testing

To demonstrate the validity and usability of LISI and to get initial estimates on the overhead introduced by using it, LISI was implemented with each of Trilinos, PETSc and SuperLU. The tests were run on the Ccaffeine [15] CCA framework. In CCA, a component corresponds to a functional decomposition, while the parallelism comes from a domain decomposition. So one component can span multiple processors, and all of its instances are called *cohorts*. Figure 3 is a rough schematic of the design of experiment which has two parts:

[a] A parallel mesh data generator sets up the finite difference operator matrix for 5-point centered differences on the unit square, with Dirichlet boundary conditions given around the boundary. The generator solves the general linear PDE

$$u_{xx} + u_{yy} - 3u_x = f$$

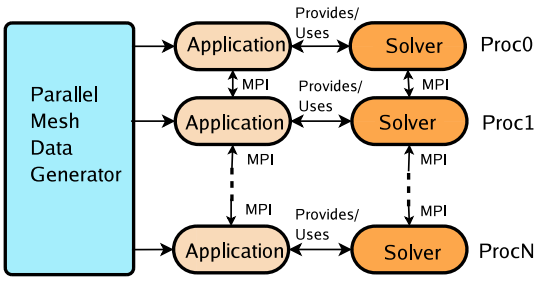

Figure 3. Test architecture

where f is function of x and y. In our test,

$$f = (2 - 6 * x - x * x) * sin(x)$$

The coefficient matrix A, RHS vector b, and solution vector x are divided conformal into block rows, one per processor. Mesh data files are written out on each compute node locally for faster data input.

[b] Solve the linear system in parallel using standard SPMD programming. The application component interacts with the solver component on each process through *provides/uses* interface and components interact with their own cohorts using the Message Passing Interface (MPI)[24] or an equivalent message-passing mechanism. The application component sends the data and user specified solver parameters to the solver component through the proposed interface, the solver component solves the given linear system, and returns its own portion of the overall solution vector back to local application component.

The test was conducted on a Linux cluster with 128 compute nodes, each with dual AMD 2.0 Ghz Opetron processor and 4 Gbytes of memory. Figure 4 shows how solver components can be switched over with the same driver component. In practice, only one of three links would show up in the component diagram. Because the testing was primarily to validate LISI and provide reference implementations, tests were run on only 1, 2, 4 and 8 processors with a coefficient matrix having 199200 non-zeros. The timing results are collected for ten runs for each experiment and a mean value is picked. Figure 5 shows the comparison of CCA component execution time (o line) with NonCCA component execution time (+ line) for PETSc, Trilinos and SuperLU solver respectively. The overhead introduced is so small that two curves are almost overlaid on each other. This demonstrates low performance impact when LISI interface layer is added, which is most important metric in HPC application.

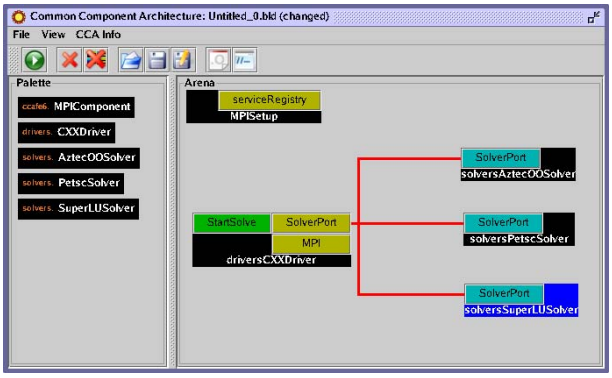

Figure 4. LISI Demo

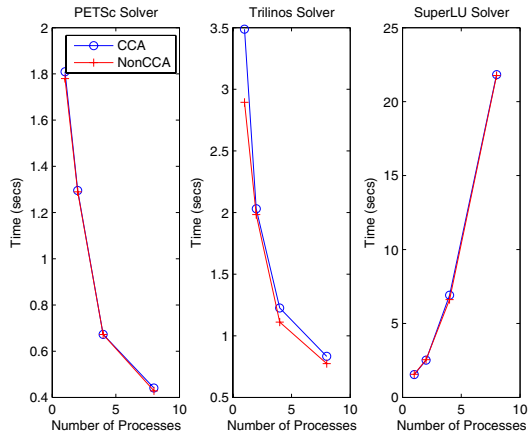

Figure 5. Comparison for PETSc, Trilinos and SuperLU components

To show the constant overhead is added by CCA framework, the PETSc solver component was tested on 8 processors with different sizes of the linear system, shown in Table 1. The fourth column in Table 1 has the difference of the second and third column and percentage of the difference by the second column. The overheads range from $0.016s$ to $0.047s$, which are very small, and are considered as constant.

The overhead percentage shows how the framework overhead affects the whole computation, and it decreases when the problem size increases. This is because we have the constant number of interface calls in the program, the overhead time would be constant no matter how large the problem size is, but with increased problem size, more computation time is typically needed both in time per iteration and the numbers of iterations.

nnz	CCA(s)	NonCCA(s)	Overhead(s)/(%)	Iters
12300	0.086	0.070	0.016/18.61	36
49600	0.189	0.144	0.045/23.73	67
199200	0.475	0.428	0.047/9.86	108
448800	1.283	1.265	0.018/1.36	165
798400	2.585	2.562	0.023/0.90	221

Table 1. Computing Times of PETSc Component with and without the LISI interface

9 Conclusion and Further Work

In this work, we have conducted investigation on widely used parallel sparse linear solver packages and tried to abstract a high level common interface among them. Designing the minimal common set of interface is not trivial, especially one needs to think across multiple packages. We examed the issues involving in the designing work, and propose the first version of LISI. We considered on how linear system characteristics, solver parameters and auxiliary data are passed through the interface; took into account variable usage scenery on how the interface is used; discussed on supporting the different data structures, parallelism, matrix-free methods, recursion and some specific requirements from CCA specification on how to choose the *uses* & *provides* ports. For easy using interface, we designed it as a single interface package, picked R-array which is more natural to the modern language programmer as array parameter type, extracted the common parameters into separated methods to avoid the potential conflict settings and had the generic parameter-setting methods to make the interface accommodate both iterative solvers and direct solvers. Prototyping and some simple experiments were done to indicate the small overhead from introducing the new layer on the existed packages. Reader may refer to the full paper [34] for design details.

We only design a top level interface for the HPC solvers, but there are still more issues that cannot be handled on this interface such as multi-level problem. And for some functionality such as matrix-free method, we only proposed the interface but haven't done real application based implementation, some updates may be introduced once it's fully implemented.

10 Acknowledgments

This work is supported in part by National Science Foundation Grants EIA-0202048, MRI CDA-0116050, and the DoE Office of Science's Center for Component Technology for Terascale Simulation Software

References

[1] Equation Solver Interface. http://z.ca.sandia.gov/esi/, 2005.

[2] All-ORders Spectral Algorithm (AORSA). (http://www.csm.ornl.gov/~shelton/fusion.html), 2006.

[3] Chasm Language Interoperability Tools. http://chasm-interop.sourceforge.net, 2006.

[4] F2PY: Fortran to Python interface generator. http://cens.ioc.ee/projects/f2py2e/, 2006.

[5] Fortran 2003 Final Committee Draft. http://www.fortran.com/fcd_announce.html, 2006.

[6] Hypre: High Performance Preconditioners. http://www.llnl.gov/CASC/linear_solvers/, 2006.

[7] Java Native Interface. http://java.sun.com/j2se/1.3/docs/guide/jni/, 2006.

[8] M3D team. http://w3.pppl.gov/~jchen, 2006.

[9] Microsoft's Common Language Runtime (CLR). http://msdn2.microsoft.com/en-us/netframework/aa497266.aspx, 2006.

[10] MUMPS: a MUltifrontal Massively Parallel sparse direct solver. http://graal.ens-lyon.fr/MUMPS/doc.html, 2006.

[11] Terascale Optimal PDE Simulation. http://www-unix.mcs.anl.gov/scidac-tops/, 2006.

[12] Tops Solver Component. http://www-unix.mcs.anl.gov/scidac-tops/solver-components/tops.html, 2006.

[13] TSTT: The Center for Terascale Simulation Tools and Technologies. http://www.tstt-scidac.org/intro/index.html, 2006.

[14] Writing C Functions in MATLAB (MEX-Files). http://cnx.org/content/m12348/latest/, 2006.

[15] B. A. Allan and R. Armstrong. Ccaffeine Framework: Composing and Debugging Applications Iteratively and Running them Statically. Compframe 2005 workshop, June 2005.

[16] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. McInnes, S. R. Parker, and B. A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC*, 1999.

[17] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[18] D. Batchelor. Integrated Simulation of Fusion Plasmas. *Physics Today*, page 35, 2005.

[19] D. M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *In 4th Annual Tcl/Tk Workshop Conference Proceedings*, July 1996. The USENIX Association, see also: http://www.swig.org.

[20] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[21] CCA-Forum. The DOE Common Component Architecture project. http://www.cca-forum.org/, 2006.

[22] V. Eijkhout. Overview of Iterative Linear System Solver Packages. *NHSE review*, 3(1), 1998. also see http://www.netlib.org/utk/papers/iterative-survey/.

[23] D. E. B. et al. CCA Distributed Array Descriptor (DAD). http://www.cca-forum.org/~data-wg/dist-array/, 2006.

[24] M. P. I. Forum. MPI: a message-passing interface standard.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[26] M. A. Heroux and etc. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software*, 31:397–423, September 2005.

[27] M. S. J. Jones and Y. Saad. Component-based Iterative Methods for Sparse Linear Systems. volume 9, pages 1–10. Concurrency and Computation: Practice and Experience, 2005.

[28] D. B. M. B. R. B. M. G. S. J. S. K. A. L. W. L. N. L. J. L. M. R. D. R. D. S. Jill Dahlburg, James Corones and H. Weitzner. Fusion Simulation Project: Integrated Simulation and Optimization of Magnetic Fusion Systems. *Journal of Fusion Energy*, 20(4):135, 2001.

[29] C. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM press.

[30] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 12-14 2001. LLNL document UCRL-JC-140349. See also http://www.llnl.gov/CASC/components/babel.html.

[31] S. L. Lee. A practical upper bound for departure from normality. 16:462–468, 1995.

[32] X. Li and J. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29:110–140, 2003.

[33] X. S. Li. An Overview of the SuperLU: Algorithms, Implementation, and user interface. *ACM Transactions on Mathematical Software*, 31:302–325, September 2005.

[34] F. C. Liu and R. Bramley. CCA-LISI: On Designing A CCA Parallel Sparse Linear Solver Interface. *Indiana University, Computer Science Department, Tech. Rep TR644*, 2007.

[35] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20:287 – 311, May 2006.

[36] B. Team. The DOE Babel Project. http://www.llnl.gov/casc/components/babel.html, 2006.

[37] D. Wesseling, Pieter. *An introduction to multigrid methods*. Pure and applied mathematics. Chichester [England]; New York: J. Wiley, 1992.

[38] Y.Saad. SPARSEKIT:A Basic Tools Kit for Sarse Matrix Computations. *Technical Report, Computer Science Department, University of Minnesota*, June 1994.