A Solver-Independent API for multi-DOF Applications using Trilinos

M.A. Heroux

Distinguished Member of the Technical Staff,
Computational Math/Algorithms Department,
Sandia National Laboratories, Albuquerque, NM, USA
E-mail: maherou@sandia.gov

Abstract: Many applications, including fully-coupled reacting flows and multiscale applications, simultaneously resolve multiple degrees of freedom (DOFs). Simultaneously, preconditioning techniques, such as segregated preconditioners for fully-coupled problems, separate variables for efficient sub-operator resolution via fast single DOF solver capabilities such as multigrid or direct solvers that would be ineffective or too expensive if applied to the full problem.

In this paper we present an application programmer interface (API) illustrating solver-neutral programming techniques via abstract interfaces, along with Trilinos-specific adapters for these interfaces. Furthermore, we show how to use Trilinos to build highly-parallel, sophisticated multi-DOF preconditioners. For concrete examples, we focus on the application Tramonto, a molecular density functional theories code used for modeling inhomogeneous fluids and EMU, a peridynamics modeling code used to predict the deformation and failure of bodies and structures and dynamic loading. However, the basic principles are easily translated to other important problem domains.

Keywords: parallel solvers; object-oriented programming; inhomogeneous fluids; Schur complement; iterative methods.

Biographical notes: Michael A. Heroux, PhD., is a Distinguished Member of the Technical Staff at Sandia National Laboratories. Prior to joining Sandia, he worked for Cray Research and Silicon Graphics, specializing parallel numerical linear algebra, large-scale parallel applications and high-performance parallel computer architecture. Presently his work involves leading the Trilinos solver project at Sandia, working on solvers for scientific and engineering applications and on future parallel computer architectures and languages.

1 INTRODUCTION

Many scientific and engineering applications have the solution of large linear systems of equations $Ax = b$ as a basic kernel, where A is a known often sparse matrix, b is a known vector and x is unknown. Many libraries are available to solve these types of problems. Some of the more popular are PETSc(Balay et al. (1998b,a, 1997)), hypre(Falgout (2006)) and Aztec(Tuminaro et al. (1999)), three collections of scalable preconditioned solvers; SuperLU(Li and Demmel (2003)), MUMPS(Amestoy et al. (2003)) and UMFPACK(Davis (2003)), three general-purpose sparse direct solver libraries; and of course LAPACK(Anderson et al. (1995)), a large set of dense solvers. Many other solver libraries are available. A comprehensive survey can be found in Dongarra and Eijkhout (2004).

In this article we focus on two aspects of integrating solver libraries into an application. First is the development of an abstract application-specific interface to solvers, which we call the *solver API*. This API is not dependent on any specific solver, but instead describes the application's view of how information will be exchanged with the solver, and what functionality is required from the solver.

The second focus of this article is a description of how to use Trilinos, Heroux (2004c), a collection of solvers with particular capabilities for multi-DOF problems, to provide sophisticated segregated preconditioned iterative methods. In particular, we show how advanced features of the Trilinos package Epetra, Heroux (2004b)–which supports parallel construction and use of distributed matrices, graphs and vectors–can make implementation of complex multi-operator preconditioned solvers straight-forward.

The remainder of this paper is organized as follows: Section 2 discusses design issues for an application-specific, solver-generic API, and illustrates such an interface by example using an interface developed for the inhomogeneous fluid modeling code Tramonto, Frink and Salinger (2000a,b). Section 3 presents some of the important classes in the Trilinos package Epetra, which supports the construction and use of parallel distributed memory linear algebra. Section 4 illustrates how specific solvers can be integrated into the solver API. Finally in Section 5 we summarize our discussion.

2 An Application-Specific, Solver-Generic Interface

Many scientific and engineering applications require the solution of an implicit system of linear equations. These applications often utilize third-party solver libraries, especially in parallel environments, since many good solvers exist that, even though considered "general-purpose," are very effective for specific types of problems. At the same time, even a single specific application often needs access to a suite of solvers to effectively address the full range of its generated problems. These facts make an application-specific, solver-generic interface attractive.

2.1 Solver API Requirements

In this section, we present and discuss five general topics related to the solver API: (i) a documented abstract interface, (ii) application-centric naming conventions, (iii) support for data reuse, (iv) granularity of abstraction and (v) a default solution capability. In our experience these topics represent the important issues the solver API must address.

Documented Abstract Interface In our experience, many if not most application-solver interfaces are buried within the application and are typically just lines of code in a larger function, and difficult to identify and isolate conceptually. One important advantage of a solver API is the ability to clearly document how a solver is used by the application. This improves code readability and maintainability, and enables easy insertion of new solver technology in the future with little or no modification to the application.

Another advantage of a documented solver API is the opportunity for solver developers to independently provide new solver capabilities that are readily used within the application. This loose coupling of the application and solver increases flexibility for all developers.

Function and Argument Naming Conventions Another important opportunity in providing an application-specific solver API is the ability to write function and argument names in terms that the application developer understands, without learning the language of solver developers. Not only does this convention improve the usability and maintainability of the API, but working solver adapters (solvers that are accessible through the API) also illustrate how application data is translated into data needed by the solver, making easier the integration of new solvers in the future. Furthermore, the exercise of developing the function and argument names for the solver API highlights the important application features that can be exploited by the solver.

Application Data Reuse Many applications solve a family of related systems of equations. For example, a nonlinear finite-element application typically generates a sequence of sparse linear systems whose matrices have a common sparsity pattern. In this case, structure-only data and computations, such as communication patterns for sparse matrix-vector multiplication in an iterative solver, can be computed once and reused by subsequent solves. In other situations, some matrix coefficient values may not change from one solve to another. Not all solvers are prepared to take advantage of such reuse, however the solver API should expose these opportunities since they represent a significant opportunity for reducing computational costs.

Granularity of Abstraction One danger of fine-grain abstraction is the runtime overhead introduced by virtual function calls and, even more importantly, the potential loss of compiler optimizations. This is not an issue for most functions in the solver API since they are either called infrequently or perform a large task when called. Initialization and solve functions are examples, respectively. However, functions that support the insertion of matrix coefficients must be carefully designed in order to avoid catastrophic runtime overheads, an issue that is well-documented (for example in Driesen and Holzle (1996)). If the solver API contains a function to insert or sum into a single matrix coefficient, then great care must be taken to design how matrix values are collected and assimilated. There is a variety of techniques to address this, including use of templates and the C++ Standard Template library Stroustrup (2000), or commonly used aggregation techniques to increase granularity.

Default General Solver The primary purpose of the solver API is to support multiple special-purpose solver capabilities. At the same time, there may be problems that do not need or have a special-purpose solver. For example, small problems or those for which no sophisticated

approach exists fall into this category. Furthermore, even when using a special-purpose solver is desired, problems of convergence or correctness of formulation may dictate that an alternative solver be available to the application. All of these issues illustrate the need for a default general solver. This solver is available via the solver API and constructs the linear system as a global problem with the matrix A stored as a single matrix object. In addition, the default general solver provides the ability to use direct methods for smaller problems, and can make debugging easier.

2.2 A Sample Solver API

To illustrate the ideas mentioned above, we introduce a specific solver API, written in C++, developed for Tramonto. Tramonto (see Frink (2006); Heroux et al. (2006a,b)) is a molecular density functional theories application used to solve a variety of inhomogeneous fluid problems. The Tramonto solver API is a good example because it exhibits all of the attributes of interest for this article and is used in production mode by Tramonto, providing Trilinos and other solvers to Tramonto users. Tramonto solves its problems over a one, two or three dimensional mesh with tens to hundreds of DOFs per mesh node. All DOFs are solved fully-coupled within a nonlinear iteration loop. However, the characteristics of each DOF can vary greatly depending on a variety of factors.

The fundamental solver API, referred to as the Basic Linear Problem Manager, with a class name of `BasicLinProbMgr`, provides both the basic solver API and concrete solution capabilities that are general purpose. The `BasicLinProbMgr` class functions are fully operational but are declared virtual so that they can be reimplemented by a derived class. We discuss one derived class: the Hard Sphere Linear Problem Manager, with the class name `HardSphereLinProbMgr`. `HardSphereLinProbMgr` inherits from `BasicLinProbMgr`, providing a reimplementation of some functions, but reusing many of the more general-purpose functions. `HardSphereLinProbMgr` is a specialized solver that takes advantage of certain structure and numerical properties in order to produce a much more robust and efficient solver (see Heroux et al. (2006a,b)).

The Basic Linear Problem Manager has seven types of functions:

1. Construction/Destruction (Table 1): Simple functions to create and delete an instance of `BasicLinProbMgr`. As the single constructor argument might suggest, Tramonto uses a single-program-multiple-data (SPMD) parallel model, implemented using the Message Passing Interface (MPI) Snir et al. (1998). Functions are:

2. Block Structure setup (Table 2): Information about number of DOFs per node, which nodes are owned by the calling processor, and additionally which are "ghost" nodes. The specific functions are:

3. Insertion functions (Table 3): Matrix and vector values that the application computes and submits for

`BasicLinProbMgr(MPI_Comm comm)`
This MPI communicator will be used by the solver manager and libraries.
`virtual  BasicLinProbMgr()`
Simple destructor.

Table 1: Constructor/Destructor

`setNumDofPerNode(int numDofPerNode)`
Number of degrees of freedom per node.
`setOwnedNodeList(int numOwnedNodes, int * ownedNodeList)`
List of Node IDs owned by the calling processor.
`setBoxNodeList(int numBoxNodes, int * boxNodeList)`
Tramonto terminology for the nodes needed for stencil operations, etc., includes owned nodes and "ghost" nodes.
`virtual finalizeBlockStructure()`
Signals to BasicLinProbMgr that submission of structural information is done and that any required preprocessing can be completed. This function is virtual because derived classes such as HardSphereLinProbMgr will want to perform the structure setup differently.

Table 2: Block Structure Setup

inserting or summing into the global linear system. The term PhysicsID, used as a root word in several arguments lists, is a Tramonto term for the labels that identify each DOF in the model. All of these functions are virtual because derived classes such as `HardSphereLinProbMgr` will want to perform the value setup differently. Also, note that there are several `virtual insertMatrixValue()` functions, each supporting a particular convenient way for Tramonto to insert matrix coefficients.

4. Linear solver functions (Table 4): There are three functions needed to solve the linear problem, shown in Table 4.

5. Nonlinear solver support functions (Table 5): Nonlinearities are very difficult to solve for some classes of Tramonto problems. To address this, continuation methods are used from the Trilinos LOCA package (see Salinger et al. (2001)). In addition to the linear solver functions, the following functions are needed by LOCA to abstractly support continuation. Note that Tramonto stores vector data types as an array of pointers, e.g., `double **b`, where `b[i]` is the array of right-hand-side values associated with the `i`'th DOF.

6. Extraction functions (Table 6): Since the control of linear problem construction is taken over by the problem manager, the application needs functions to access coefficient information, primarily for diagnostics and the ability to modify existing values.

virtual initializeProblemValues()
Function that must be called each time prior to starting matrix, lhs and rhs value insertion (usually once per nonlinear iteration). This function indicates that non-structural data should be reset.
`virtual insertRhsValue(int ownedPhysicsID, int ownedNode, double value)`
Insert rhs value based on ownedNode and ownedPhysicsID.
`virtual insertMatrixValue (int ownedPhysicsID, int ownedNode, int boxPhysicsID, int boxNode, double value)`
Insert single matrix coefficient into system.
`virtual insertMatrixValues (int ownedPhysicsID, int ownedNode, int boxPhysicsID, int *boxNodeList, double *values, int numEntries)`
Insert matrix coefficients for a given row, where columns are all from the same physics type at different nodes.
`virtual insertMatrixValues (int ownedPhysicsID, int ownedNode, int *boxPhysicsIDList, int boxNode, double *values, int numEntries)`
Insert matrix coefficients for a given row, where columns are from different physics types at the same node.
`virtual finalizeProblemValues()`
*Signals to **BasicLinProbMgr** that submission of value information is done and that any required preprocessing can be completed.*

Table 3: Insertion Functions

`virtual setupSolver(int *solverOptions, double *solverParams)`
Pass options and parameters to the solver setup. This function does any necessary preprocessing to make the solver more efficient.
`virtual solve()`
Actual solver invocation. Requires no parameters.
`virtual getSolverStatus(double *solverStatus)`
*After the **solve()** function has completed, this function allows the application to get more detailed information about the success of the solution process.*

Table 4: Linear Solver Functions

7. Support functions (Table 7): These functions help for debugging and for external analysis. The first function writes the matrix out to an ASCII text file in the Matrix-Market format, a file with three columns where each line of the file contains the row, column and value for a matrix coefficient, R. Pozo (2006). The right-hand-side and left-hand-side are written out

`virtual setRhs(double **b)`
Set all right hand side vectors at once.
`virtual setLhs(double **x)`
Set all left hand side vectors at once.
`virtual applyMatrix(const double **x, double **b)`
Apply global linear operator, b = Ax.
`virtual importR2C(const double **xOwned, double **xBox)`
Fill the arrays xBox[i] with xOwned[i] for all physics types i, i.e., fill in ghost values on each processor.

Table 5: Nonlinear Solver Support Functions

`virtual double getMatrixValue (int ownedPhysicsID, int ownedNode, int boxPhysicsID, int boxNode)`
Get a matrix entry.
`virtual getRhs(double **b)`
Get all right hand side vectors at once.
`virtual getLhs(double **x)`
Get all left hand side vectors at once.

Table 6: Extraction Functions

as an ASCII file of floating point numbers, compatible with MATLAB, The Mathworks (2006).

`virtual writeMatrix(const char *filename, const char *matrixName, const char *matrixDescription`
Write matrix to specified filename using Matrix Market (i,j,value) format.
`virtual writeRhs(const char *filename)`
Write right hand side to specified filename in Matlab-compatible format.
`virtual writeLhs(const char *filename)`
Write left hand side to specified filename in Matlab-compatible format.

Table 7: Support Functions

2.3 Special-purpose Solver API: Hard Spheres

The solver API discussed in Section 2.2 is specific to Tramonto, but general purpose from the solver perspective. In our work with Tramonto, we have developed very specific algorithms for certain classes of problems. One such class of problems is hard spheres. Hard sphere models are used to predict the density of atomic fluids near surfaces. Accurate density profiles require a nonlocal density model such that the density at each node near a surface is a function of nearby densities. Nonlocal density variables are auxiliary to the problem, but are retained for easier construction of the coefficient matrix of the global problem. For a given problem with N grid points, a sample one dimensional hard sphere problem will have $11N$ unknowns,

4

i.e., 11 DOFs per node. However, 10 of these DOFs are associated with nonlocal densities while only one is associated with the primitive density at each node. Furthermore, the relationship between the nonlocal densities themselves is very simple, if properly ordered.

The block structure of a hard sphere matrix after proper reordering is:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{1}$$

where

$$A_{11} = \begin{pmatrix} I & 0 \\ X & I \end{pmatrix}. \tag{2}$$

In our example, A_{11} is of dimension $10N$ and A_{22} of dimension N. Immediately we see that

$$A_{11}^{-1} = \begin{pmatrix} I & 0 \\ -X & I \end{pmatrix}. \tag{3}$$

Thus, the Schur complement S of A with respect to A_{22} is $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ and is easily computed since A_{11}^{-1} is readily constructed. Therefore, by proper ordering, we can efficiently and directly eliminate $10N$ variables, leaving only the Schur complement system to be solved by some other implicit solver.

Without going into further details of the algorithms used to solve hard sphere problems, which can be learned from Heroux et al. (2006a,b), we can immediately see that any solver wishing to take advantage of the special behavior of nonlocal densities must first learn, from the application, which DOFs are associated with the nonlocal densities. This information must be given prior to submission of matrix entries and is part of the Type 2 functions in Section 2.2.

One of the key design features of object-oriented programming is inheritance, the ability of one class to inherit the functionality and data members from another. We use inheritance in the solver APIs as follows: `BasicLinProbMgr` provides an implementation of all functions described in Section 2.2. `HardSphereLinProbMgr` inherits from `BasicLinProbMgr` and therefore automatically inherits all of the functions and data of `BasicLinProbMgr`. In addition, `HardSphereLinProbMgr` defines several new functions that allow the application to provide the special information needed to identify nonlocal density DOFs. Furthermore, `HardSphereLinProbMgr` redefines several functions that were previously defined by `BasicLinProbMgr` in order to take advantage of the special structure found in hard sphere problems.

Specifically, `HardSphereLinProbMgr` provides the three Type 2 functions found in Table 8. In addition, the `HardSphereLinProbMgr` class redefines the functions: `finalizeBlockStructure()`, `initializeProblemValues()`, `insertMatrixValue()`, `finalizeProblemValues()`, `setupSolver()`, `solve()`, `writeMatrix()` and `applyMatrix()`. These functions have the same argument list as in `BasicLinProbMgr` but now function differently to take advantage of the special structure found in hard sphere problems.

`setIndNonLocalDensityIDs(int numIndNonLocalDensity, int *physicsIDs)` *Define independent non-local density (upper block of A_{11}).*
`setDepNonLocalDensityIDs(int numDepNonLocalDensity, int *physicsIDs)` *Define dependent non-local density (lower block of A_{11}).*
`setPrimitiveDensityIDs(int numPrimitiveDensity, int *physicsIDs)` *Define primitive density (A_{22} block).*

Table 8: `HardSphereLinProbMgr`-specific Functions

It is worth noting that only a few lines of application code differ between using the `BasicLinProbMgr` and `HardSphereLinProbMgr`. Specifically these differences occur when the `HardSphereLinProbMgr` constructor is called and when the three special functions are called to define the nonlocal and primitive densities. All of these functions are called once in the very beginning of the application run. After these setup steps, the application will use a `BasicLinProbMgr` and `HardSphereLinProbMgr` identically through the same function calls.

2.4 Further Details of the Tramonto Interface

Tramonto is written in C. As a result, although the API we described is object-oriented and written in C++, we provide a simple wrapper layer to minimize the impact on Tramonto. Since C does not support object-orientation directly, we create a set of functions as illustrated here:

```
void * dft_basic_lin_prob_mgr_create(MPI_Comm comm) {
  dft_BasicLinProbMgr * linprobmgr_ =
    new dft_BasicLinProbMgr(comm);
  return((void *)linprobmgr_);
}

void dft_linprobmgr_destruct(void * linprobmgr) {
  dft_BasicLinProbMgr * linprobmgr_ =
    (dft_BasicLinProbMgr *) linprobmgr;
  delete linprobmgr_;
}

int dft_linprobmgr_setnumdofpernode
    (void * linprobmgr, int numdofpernode) {
  dft_BasicLinProbMgr * linprobmgr_ =
    (dft_BasicLinProbMgr *) linprobmgr;
  return(linprobmgr_->setNumDofPerNode(numdofpernode));
}
```

The first function listed above creates a new instance of the `BasicLinProbMgr` class. Then the function casts the pointer to `void *` and returns it to Tramonto. The destructor and other functions accept the `linprobmgr` pointer as first argument, immediately casting it back to the correct class type. Using this approach, strong type-checking is lost for the class instantiation because of the object cast-to-void-pointer step. However, this approach allows Tramonto functions to remain as C implementations and make the use of C++ more transparent. Note that, other than constructors and destructors, the solver API functions return integer codes. These codes are used to

5

pass error conditions back to Tramonto. The exact values returned are documented in the solver API.

Use of the solver API in Tramonto has greatly simplified both the use of a general solver, which was the only solver capability originally available in Tramonto, and the introduction of new specialized solvers. Because all solver data is passed between the application and solver via the solver API, it is very clear which parts of the application deal with solver data. Furthermore, introducing a new specialized solver requires only a few extra lines of code in Tramonto, needed to (i) instantiate the specialized problem manager instead of the basic problem manager and (ii) pass any additional information needed by the specialized problem manager. All other solver API calls remain unchanged. The following code fragment shows how Tramonto instantiates either a `BasicLinProbMgr` or `HardSphereLinProbMgr` instance:

```
if (Type_func != HS) {
  /* Build arrays indnonlocaleq, depnonlocaleq, densityeq arrays
     containing PhysicIDs of the independent and dependent
     nonlocal density, and primitive density DOFs, respectively.
     These are simply loops executed one time.  Omitted here for
     brevity. */

  /* Create an instance of a Hard Sphere Linear Problem manager */
  LinProbMgr_manager = dft_hardsphere_lin_prob_mgr_create
      (MPI_COMM_WORLD);
  dft_hardsphere_lin_prob_mgr_setindnonlocalequationids
      (LinProbMgr_manager, Nindnonlocal, indnonlocaleq);
  dft_hardsphere_lin_prob_mgr_setdepnonlocalequationids
      (LinProbMgr_manager, Ndepnonlocal, depnonlocaleq);
  dft_hardsphere_lin_prob_mgr_setdensityequationids
      (LinProbMgr_manager, Ndensity, densityeq);
}
else { /* Default is basic manager */
  LinProbMgr_manager = dft_basic_lin_prob_mgr_create(MPI_COMM_WORLD);
}
/* This call and all remaining solver API calls are identical
   regardless of which problem manager was instantiated. */
dft_linprobmgr_setnumdofpernode(LinProbMgr_manager, numPhysicsIDs);
```

After our initial development of a specialized Hard Sphere Linear Problem Manager, we continued algorithms research and development for Tramonto and developed new specialized problem managers. Without going into detail, we next developed a specialized manager for polymer chain problems. These problem have a very complex block structure with tens to hundreds of DOFs per node. The details can be found in Heroux et al. (2006a,b). What is worth noting here is that introducing the new specialized problem manager into Tramonto required on the following additional code fragment and no other changes, inserted just before the "else" statement in the previous code fragment:

```
else if (Type_poly == CMS) {

  /* Build arrays geq, ginveq, cmseq, and densityeq arrays
     containing PhysicIDs of the "G", "G inverse", "CMS field" and
     primitive density DOFs, respectively.
     These are simply loops executed one time.  Omitted here for
     brevity. */

  /* Create an instance of a Polymer Linear Problem manager */
  LinProbMgr_manager = dft_poly_lin_prob_mgr_create(MPI_COMM_WORLD);
  /* Define DOF information needed by this manager */
  dft_poly_lin_prob_mgr_setgequationids
      (LinProbMgr_manager, Ng, geq);
  dft_poly_lin_prob_mgr_setginvequationids
      (LinProbMgr_manager, Nginv, ginveq);
  dft_poly_lin_prob_mgr_setcmsequationids
      (LinProbMgr_manager, Ncms, cmseq);
```

```
  dft_poly_lin_prob_mgr_setdensityequationids
      (LinProbMgr_manager, Ndensity, densityeq);
}
```

2.5 Solver APIs for Other Applications

Although the solver API discussed in Sections 2.2 and 2.3 is very specifically written for Tramonto, it can be easily understood what would differ when developing a solver API for another application. We have used this same basic approach for EMU Silling (2006), a Fortran 95 peridynamics modeling code used to predict the deformation and failure of bodies and structures and dynamic loading. The EMU solver API is remarkably similar to Tramonto's. The basic seven types of functions are very similar, although EMU does not presently need continuation methods or the corresponding Type 5 functions. The Type 2 functions are essentially identical. Type 3 functions are different because of how EMU generates its nonzero coefficients, but the differences are only in argument names and organization of the arrays. Type 4 functions, related to configuring and calling the solver are quite similar. Type 6 and 7 functions, extraction and write functions, are also very similar. Of course, in all of these functions the terminology changes to match that of EMU.

3 Epetra Concepts and AztecOO

Prior to discussing specific solvers that implement the solver API of Section 2, we must first discuss some features of the Trilinos package Epetra (Heroux (2004b)). Epetra is a collection of linear algebra classes that support the construction and use of vectors, multivectors, graphs, matrices and linear operators. Epetra is intended for parallel, distributed memory architectures and supports the construction of distributed memory objects in parallel with minimal inconvenience for the user, by which we mean that Epetra insolates the user from as many details of parallel distributed memory computations as possible with the constraint that performance not be greatly impacted. Furthermore, Epetra has a very flexible data distribution capability that supports placement of vector, graph and matrix entries anywhere on the parallel machine. This capability is especially useful in the context of multi-physics applications.

In this section we proceed with a bottom-up approach, introducing in order `Epetra_Comm`, `Epetra_Map`, `Epetra_Vector`, `Epetra_MultiVector`, `Epetra_CrsGraph`, `Epetra_CrsMatrix`, `Epetra_RowMatrix` and finally `Epetra_Operator`. We also mention `AztecOO`, since its use illustrates the role of the implicit linear solver.

3.1 Epetra_Comm

Epetra does not directly depend on the MPI parallel programming model *a priori*. Instead it uses an abstract class called `Epetra_Comm` to encapsulate parallel machine details. `Epetra_SerialComm` and `Epetra_MpiComm` inherit

from the base `Epetra_Comm` class and provide serial and MPI implementations of the `Epetra_Comm` interface, respectively. An `Epetra_Comm` instance is needed to create most other Epetra objects and the functions found in `Epetra_Comm` are very similar to MPI functions. Epetra is strongly a single-program-multiple-data (SPMD) parallel programming model. `Epetra_Comm` reflects this fact.

3.2 Epetra_Map

The `Epetra_Map` class encodes information about the layout of linear algebra objects on the parallel machine. The primary content of an `Epetra_Map` on each processor is a list of global IDs (signed integers called GIDs) that are in some way managed by the processor. The standard `Epetra_Map` of 100 GIDs from 0 to 99 on three processors would put GIDs 0 through 33 on PE 0, 34 through 66 on PE 1, and 67 through 99 on PE 3. This `Epetra_Map` would be constructed with the following C++ statement, run as part of a three-processor MPI job:

```
Epetra_MpiComm comm( MPI_COMM_WORLD );
Epetra_Map map(100, 0, comm);
```

Beyond this simple use case, `Epetra_Maps` can be much more complex. In particular, the list of GIDs assigned to each processor does not need to be contiguous or start at zero, nor do GIDs need to be uniquely owned. Both of these features are important for multi-physics problems. For future reference, we refer to an `Epetra_Map` whose GIDs appear only once in the map, i.e., are not repeated anywhere on any processor, as ***1-to-1 maps***.

3.3 Epetra_Vector

The `Epetra_Vector` class supports the construction and use of distributed dense vectors. An `Epetra_Vector` is constructed by using an existing `Epetra_Map` argument:

```
Epetra_MpiComm comm( MPI_COMM_WORLD );
Epetra_Map map(100, 0, comm);
Epetra_Vector x(map);
```

The `Epetra_Vector` constructed above will have the same size on each processor as the `Epetra_Map` object used to construct it. Furthermore, the GIDs in the `Epetra_Map` are associated with the corresponding entries of the `Epetra_Vector`.

3.4 Epetra_MultiVector

The `Epetra_MultiVector` class is very similar to the `Epetra_Vector` and supports the construction and use of a collection of distributed dense vectors all with the same length. An `Epetra_MultiVector` is constructed by using an existing `Epetra_Map` argument and an integer declaring how many vectors are part of the multivector:

```
Epetra_MpiComm comm( MPI_COMM_WORLD );
Epetra_Map map(100, 0, comm);
Epetra_Vector x(map, 5);
```

The `Epetra_MultiVector` constructed above will have the same size on each processor as the `Epetra_Map` object used to construct it and contain 5 `Epetra_Vectors`. Furthermore, the GIDs in the `Epetra_Map` are associated with the corresponding rows of the `Epetra_MultiVector`.

3.5 Epetra_CrsGraph

The `Epetra_CrsGraph` class supports the construction of the nonzero pattern or "skeleton" of one or more sparse matrices. The "Crs" signifies a compressed row storage bias such that entries in the graph are stored contiguously by row. Of course, the transpose is implicitly column oriented, so it is possible to support column oriented operations with the same class. The importance of the `Epetra_CrsGraph` class is that it supports the one-time construction of structural information that can used across multiple Epetra matrices or across multiple uses of the same Epetra matrix where the values change from use to use. Both situations are common and important.

A `Epetra_CrsGraph` object has four `Epetra_Map` objects associated with it:

rowMap On each processor, the GIDs of the rows that will be managed at least in part by the processor. A rowMap is typically 1-to-1, meaning that GIDs appear only once in the map.

colMap On each processor, the GIDs of the columns that will be managed at least in part by the processor. A colMap is typically *not* 1-to-1 when constructed on more than one processor.

domainMap On each processor, the distribution of `Epetra_Vector` and `Epetra_MultiVector` objects that will be used as domain vectors for matrices built using this `Epetra_CrsGraph` object. Note that this `Epetra_Map` *must* be 1-to-1. It is also often the same as the rowMap, but not always.

rangeMap On each processor, the distribution of `Epetra_Vector` and `Epetra_MultiVector` objects that will be used as range vectors for matrices built using this `Epetra_CrsGraph` object. Note that this `Epetra_Map` *must* be 1-to-1. It is also often the same as the rowMap and domainMap but not always.

3.6 Epetra_CrsMatrix

Related to the `Epetra_CrsGraph` is the `Epetra_CrsMatrix`. This class supports the construction of sparse matrices with a row-oriented bias such that matrix entries are stored contiguously by row. `Epetra_CrsMatrix` objects can be constructed by passing in an existing `Epetra_CrsGraph` object, and then filled against the pre-determined graph structure. Or the `Epetra_CrsGraph` can be implicitly constructed "on-the-fly" as part of the matrix value insertion process. An `Epetra_CrsMatrix` object has the same four maps as an `Epetra_CrsGraph`, with the same meaning. Continuing to elaborate on the concepts of the four maps,

let us consider the following 8-by-8 matrix:

$$
A = \left(
\begin{array}{cc|cc|cc|cc}
8 & -3 & -2 & -1 & 0 & 0 & 0 & 0 \\
-4 & 9 & -4 & -3 & 0 & 0 & 0 & 0 \\
\hline
-2 & -3 & 8 & -3 & -2 & -1 & 0 & 0 \\
-2 & -3 & -4 & 9 & -4 & -3 & 0 & 0 \\
\hline
0 & 0 & -2 & -3 & 8 & -3 & -2 & -1 \\
0 & 0 & -2 & -3 & -4 & 9 & -4 & -3 \\
\hline
0 & 0 & 0 & 0 & -2 & -3 & 8 & -3 \\
0 & 0 & 0 & 0 & -2 & -3 & -4 & 9
\end{array}
\right). \quad (4)
$$

Standard Matrix and Vector Distribution A standard distribution of this matrix on two processors (2 PEs) would give all entries for the first four rows to PE 0 and all entries for the last four rows to PE 1. In the standard case, we would also partition any vectors used with this matrix, e.g., for computing $y = Ax$, using rowMap as the rangeMap (the map for y) and as the domainMap (the map for x). Thus, rowMap, colMap, domainMap and rangeMap would be:

	PE 0	PE 1
rowMap	$\{0,1,2,3\}$	$\{4,5,6,7\}$
colMap	$\{0,1,2,3,4,5\}$	$\{4,5,6,7,2,3\}$
domainMap	$\{0,1,2,3\}$	$\{4,5,6,7\}$
rangeMap	$\{0,1,2,3\}$	$\{4,5,6,7\}$

A pseudo-code fragment to construct this matrix with the standard distribution is as follows:

```
Epetra_MpiComm comm( MPI_COMM_WORLD );
if (comm.MyPID()==0)
    rowMapGids[] = {0, 1, 2, 3};
else
    rowMapGids[] = {4, 5, 6, 7};

Epetra_Map map(-1, rowMapGids, 0, comm);

Epetra_CrsMatrix A(Copy, map, 0);
for (int i=0; i<map.NumMyElements(); i++)
    // Insert values/indices for a row
    // (details omitted)
    A.InsertGlobalValues(rowMapGids[i], ...)
// Signal completion of Insert process.
A.FillComplete();  // Matrix  now ready to use.
```

Note that in the above code fragment, when `A.FillComplete()` is executed, colMap is implicitly constructed using the set of column IDs submitted, and domainMap and rangeMap are implicitly set equal to rowMap.

Now suppose that we happen to want a different layout for the domain and range vectors and define them to be:

	PE 0	PE 1
domainMap	$\{0,2,4,6\}$	$\{1,3,5,7\}$
rangeMap	$\{1,3,5,7\}$	$\{0,2,4,6\}$

In this case, we replace the call to `A.FillComplete()` with `A.FillComplete(domainMap, rangeMap)`. As long as the domainMap contains a single instance of each GID in the colMap and rangeMap contain a single instance of each GID in the rowMap, the call to `A.FillComplete(domainMap, rangeMap)` will construct a communication pattern to perform parallel matrix computations involving `A`.

Block Matrix and Vector Distribution The matrix A in Equation 4 was designed to mimic a 4-node, 2-DOF problem, as indicated by the partitioning. The ordering of A in Equation 4 is considered node-first, i.e., all equations associated with a node are listed first. An alternative ordering, more amenable to segregated preconditioning is to order equations DOF-first, followed by partitioning the DOFs into separate matrices. In this case the matrix is effectively permuted to:

$$
\tilde{A} = \left(
\begin{array}{cccc|cccc}
8 & -2 & 0 & 0 & -3 & -1 & 0 & 0 \\
-2 & 8 & -2 & 0 & -3 & -3 & -1 & 0 \\
0 & -2 & 8 & -2 & 0 & -3 & -3 & -1 \\
0 & 0 & -2 & 8 & 0 & 0 & -3 & -3 \\
\hline
-4 & -4 & 0 & 0 & 9 & -3 & 0 & 0 \\
-2 & -4 & -4 & 0 & -3 & 9 & -3 & 0 \\
0 & -2 & -4 & -4 & 0 & -3 & 9 & -3 \\
0 & 0 & -2 & -4 & 0 & 0 & -3 & 9
\end{array}
\right). \quad (5)
$$

Keeping the global IDs the same as in Equation 4, a pseudo-code fragment to construct this matrix collected in four submatrices is as follows:

```
Epetra_MpiComm comm( MPI_COMM_WORLD );

if (comm.MyPID()==0) {
    rowMapGids1[] = {0, 2};
    rowMapGids2[] = {1, 3}; }
else {
    rowMapGids1[] = {4, 6};
    rowMapGids2[] = {5, 7}; }

Epetra_Map map1(-1, rowMapGids1, 0, comm);
Epetra_Map map2(-1, rowMapGids2, 0, comm);

Epetra_CrsMatrix A11(Copy, map1, 0);
Epetra_CrsMatrix A12(Copy, map1, 0);
Epetra_CrsMatrix A21(Copy, map2, 0);
Epetra_CrsMatrix A22(Copy, map2, 0);
for (int i=0; i<map.NumMyElements(); i++) {
    // Insert values/indices for a row
    // Put into A11, A12, A21 or A22
    // as appropriate (details omitted)
    A11.InsertGlobalValues(rowMapGids[i], ...) }
// Signal completion of Insert process.
// Matrices  now ready to use.

A11.FillComplete(); // Same as A11.FillComplete(map1,map1)
A12.FillComplete(map2, map1);
A21.FillComplete(map1, map2);
A22.FillComplete(); // Same as A22.FillComplete(map2,map2)
```

There are several things worth noting in this example. First, the order in which global matrix entries are processed remains the same as in the previous example. This is important from an application perspective since the application needs to be free from details such as how matrix entries are being stored. Second, the importance of the generality of the `Epetra_Map` class, specifically the ability to use arbitrary integer values to define GIDs and the ability to specify the domain and range map with a different GID set than the row or column map, becomes apparent with this example. Finally, the `Epetra_CrsMatrix` object A11, A12, A21 and A22 are all distributed across the parallel machine such that, if the global `Epetra_CrsMatrix` object A were well-distributed, so are these submatrices.

The ability to create multiple matrices with a common GID space is very important for ease-of-use in multiphysics applications, because we need to view the problem

from two levels: (i) the global linear system level which is the full problem we need to solve and (ii) the sub-problem level where specialized solver algorithms work on smaller pieces of the problem. The common GID space supports exchange of problem data back-and-forth between these two levels using data transfer functions in Epetra that greatly simplify the programming effort. In the case of Tramonto and the `PolyLinProbMgr` class, the preconditioner that is most effective requires the construction of dozens to hundreds of distributed `Epetra_CrsMatrix` objects. That these `Epetra_CrsMatrix` objects share a GID space is very important.

There are many Epetra classes that we do not discuss here. In particular there are several other sparse matrix classes that may be more appropriate for some applications than `Epetra_CrsMatrix`. In particular, the `Epetra_VbrMatrix`, `Epetra_FECrsMatrix` and `Epetra_FEVbrMatrix` classes are appropriate in many situations. For more details about these classes, and Epetra performance issues in general, please consult the Epetra Performance Optimization Guide, Heroux (2005).

The remaining two Epetra classes we will discuss here are `Epetra_Operator` and `Epetra_RowMatrix`. Both of these classes are *pure virtual classes*, meaning they have no functional code.

3.7 `Epetra_RowMatrix`

`Epetra_RowMatrix` is an interface to all Epetra sparse matrix classes including `Epetra_CrsMatrix`, `Epetra_VbrMatrix`, `Epetra_FECrsMatrix` and `Epetra_FEVbrMatrix`. A class that is a user of sparse matrices, such as a preconditioner class, does not typically need to know the details of how a sparse matrix is stored. Instead access is needed to matrix entries in a reasonably efficient way. `Epetra_RowMatrix` provides this kind of read-only capability for any class that inherits its interface. In particular, `Epetra_RowMatrix` provides a single function that, when called with a specified row index, must fill arrays with values and indices from that row of the matrix. All preconditioners in Trilinos are implemented using this basic concept. Any matrix class that can provide a row of matrix coefficients on demand can easily inherit from the `Epetra_RowMatrix` interface.

3.8 `Epetra_Operator`

`Epetra_Operator` is an even simpler interface than `Epetra_RowMatrix`. It provides an Apply() function or an ApplyInverse() function, or both, depending on the concrete class that inherits its interface. All Epetra distributed matrix classes inherit from the `Epetra_Operator` interface because `Epetra_RowMatrix` inherits from `Epetra_Operator`. Furthermore, many preconditioners in Trilinos inherit from `Epetra_Operator`, e.g., the algebraic preconditioners in IFPACK, Heroux and Sala (2004), and the multi-level preconditioners in ML, Tuminaro and Hu (2004). We will revisit `Epetra_Operator` in

Section 4 when we discuss implementing Schur complement operators.

3.9 AztecOO

The Trilinos package AztecOO, Heroux (2004a), contains a class of the same name, `AztecOO`. This class provides access to a variety of preconditioners and iterative solvers for linear systems of the form $Ax = b$. In addition `AztecOO` provides support for using Epetra objects as matrices, linear operators and vectors to define the linear problem for `AztecOO`. The arguments x and b can be `Epetra_Vector` or `Epetra_MultiVector` objects. The linear operator A can be an `Epetra_RowMatrix` or simply an `Epetra_Operator`. Furthermore, `AztecOO` accepts a user-built preconditioner as an `Epetra_Operator`. When `AztecOO` needs to apply A to a vector, it calls the Apply() function. When it needs to apply the preconditioner to a vector it calls the ApplyInverse() function. In this way, `AztecOO` supports the use of very complex formulations of A and its preconditioners. In particular, `AztecOO` can be used to solve Schur complement systems where A takes the form of $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$, as in Section 2.3. S need not be formed, but supplied via the Apply() function of an `Epetra_Operator` class. Similarly the preconditioner for S can be provided by the ApplyInverse() function of the same `Epetra_Operator` class.

4 Concrete Solver Implementations

In Section 2 we presented the important features of a solver API and the particular example of Tramonto to illustrate our ideas. In Section 3 we discussed important Epetra classes and how these classes can be used to accept matrix coefficients from the application for insertion into a single global matrix, or separated into several submatrices as would be needed for some kinds of segregated preconditioners. In this section we discuss the complement to the solver API: the concrete solver implementations. The primary purpose of this section is to show how information that is passed to the linear problem manager via the solver API is parsed for use by the solver. Unlike Section 2, where discussion was very application-specific, in this section data and terminology will be very solver-specific, using the classes discussed in Section 3.

4.1 The `Schur_Epetra_Operator` Class

To drive this discussion, we consider the hard sphere problem presented in Section 2.3. The solution strategy for this problem is to solve the Schur complement system, using GMRES from `AztecOO` and taking advantage of the fact that A_{11}^{-1} can be explicitly formed. In order to do this, we first introduce a new class called `Schur_Epetra_Operator`. This class accepts four arguments to its constructor:

A11 An `Epetra_Operator` representing the A_{11} block of the matrix in Equation 1. This `Epetra_Operator` must provide an implementation of both the Apply()

and ApplyInverse() functions. Note that the Apply-Inverse() function will be used most often as part of applying $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$. Since A_{11}^{-1} can be explicitly formed, we store A_{11}^{-1} which, as indicated by Equation 3, mean negating the off-diagonal terms. It is also worth noting that the diagonal entries need not be kept since they are known to be ones.

A12 An `Epetra_CrsMatrix` containing the coefficients of the A_{12} block of Equation 1.

A21 An `Epetra_CrsMatrix` containing the coefficients of the A_{21} block of Equation 1.

A22 An `Epetra_Operator` representing the A_{22} block of the matrix in Equation 1. This `Epetra_Operator` must provide an implementation of both the Apply() and ApplyInverse() functions. However, the Apply-Inverse() function need not be exact. It will be used by `Schur_Epetra_Operator` as part or all of the pre-conditioner for the Schur complement system. In this particular example, ApplyInverse() will apply the inverse of the diagonal of A_{22}.

Since `Schur_Epetra_Operator` is an `Epetra_Operator`, and will be used with `AztecOO` to solve the Schur complement system, `Schur_Epetra_Operator` must provide an implementation of both the Apply() and ApplyInverse() functions. The Apply() function must supply the action of $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ on an `Epetra_Vector` or `Epetra_MultiVector`. The ApplyInverse() function will be used to supply the preconditioner, so need only be an approximation to S^{-1}. In our case we approximate S^{-1} by first assuming $S \approx A_{22}$ and then using Jacobi (diagonal) scaling of A_{22}. Thus, $S^{-1} \approx (diag(A_{22}))^{-1}$.

4.2 Helpful Utilities

Writing complex linear operators such as `Schur_Epetra_Operator` requires handling many data objects. Most of these object must persist outside of the immediate scope in which they were created. To facilitate managing creation, but especially deletion of these objects, we use the concept of a *smart pointer*. The Boost C++ Libraries (see Boost.org (2006)) provide such a class. However, the Trilinos package Teuchos, Thornquist et al. (2004), provides a Boost-compatible smart pointer that has a few additional useful features. The `Teuchos::RefCountPtr` class supports the declaration of a templated-type pointer that manage reference counts and deletion of objects. Of course, languages such as Java and Python automatically handle garbage collection as part of the language, but C++, C and Fortran do not. A `Teuchos::RefCountPtr` instance can be a simple attribute of a C++ class and constructed with a null pointer, then later assigned a non-trivial pointer. Thus, the line:

```
Teuchos::RefCountPtr<Epetra_CrsMatrix> A11invMatrix_;
```

can appear in the class declaration and the following line:

```
A11invMatrix_ = Teuchos::rcp(new Epetra_CrsMatrix(Copy,
                       OperatorRangeMap(), 0));
```

can appear in the class implementation. In this way, `A11invMatrix_` can persist, be shallow-copied, and be automatically deleted, without concerns for memory leaks or access deleted memory. In addition the `->` operator is implemented by `Teuchos::RefCountPtr` such that expressions like:

```
A11invMatrix_->InsertGlobalValues(row, 1, &value, &col);
```

work the same as they would when working with a raw `Epetra_CrsMatrix` pointer.

Another useful utility in Teuchos is a set of macros that provide tests for exceptions. In particular, the macro

```
TEST_FOR_EXCEPT(bool arg);
```

is a light-weight means of testing pre and post conditions for functions. For example in the Apply() function of the `Schur_Epetra_Operator` the following preconditions must be true:

```
int Schur_Epetra_Operator::Apply(
              const Epetra_MultiVector& X,
              Epetra_MultiVector& Y) const {
TEST_FOR_EXCEPT(!X.Map().SameAs(OperatorDomainMap()));
TEST_FOR_EXCEPT(!Y.Map().SameAs(OperatorRangeMap()));
TEST_FOR_EXCEPT(Y.NumVectors()!=X.NumVectors());

// Rest of function omitted...
}
```

Both of the above utilities were used in the implementation of the Tramonto solver API. In our experience, such utilities are invaluable for producing high-quality software in complex environments.

4.3 Implementing the Hard Sphere Linear Problem Manager

At this point, we finally have all of the necessary elements to discuss the implementation of the `HardSphereLinProbMgr` class. Although the level of detail presented here should suffice for gaining understanding and using similar techniques for other applications, we omit some specific information for readability. Readers who are interested in obtaining the functioning C++ source code can contact the author directly.

The `HardSphereLinProbMgr` version of the constructor and destructor listed in Table 1 will be simple and similar to `BasicLinProbMgr`. `HardSphereLinProbMgr` will inherit the `BasicLinProbMgr` implementation of the first 3 block structure setup functions in Table 2. The functions found in Table 8, when called by the application, will provide additional structural information needed to identify which DOFs belong to which blocks in Equation 1. The collection of these Type 2 functions provides sufficient information to form the `Epetra_Map` objects for the rowMap, domainMap and rangeMap for all four arguments A11, A12, A21 and A22 needed by `Schur_Epetra_Operator`. The colMap objects will be constructed automatically by Epetra. These `Epetra_Maps` will be created when the `HardSphereLinProbMgr` version of finalizeBlockStructure() is called.

`HardSphereLinProbMgr` reimplements all of the matrix and vector insertion functions in Table 3. Thus, once the

`Epetra_Map` objects are constructed, the insertion functions accept matrix and vector values and corresponding node and physic IDs. Using the `MyGID(int GID)` function that is a member function of the `Epetra_Map` class, it is easy to determine where each matrix value should be submitted, either to the insertion functions of the `Epetra_Operator`s A11 or A22, or the insertion functions of the `Epetra_CrsMatrix` objects A12 or A21. This process will be similar to what was illustrated for the block matrix in Equation 5 in Section 3.6. Once all values are inserted, the application will call `finalizeProblemValues()` which is also reimplemented by `HardSphereLinProbMgr`. Calling this function will prompt the following actions:

1. The `finalizeProblemValues()` function on `Epetra_Operator` object A11 will be called, signaling that off-diagonal elements of A_{11} in Equation 1 (which are the only values kept, as mentioned above) stored in an `Epetra_CrsMatrix` are completely entered. Thus, the FillComplete() member function of `Epetra_CrsMatrix` will be called and the `Epetra_Operator` A11 completely constructed.

2. The `finalizeProblemValues()` function on `Epetra_Operator` object A22 will be called, signaling that all entries of A_{22} in Equation 1 are entered. A_{22} is stored in an `Epetra_CrsMatrix`. In addition the reciprocal of $diag(A_{22})$ is kept for use as part of the preconditioner for S.

3. The FillComplete() member function of `Epetra_CrsMatrix` will be called for both `Epetra_CrsMatrix` objects A12 and A21.

4. The constructor for `Schur_Epetra_Operator` will be called, passing A11, A12, A21 and A22 in as arguments.

After these steps are complete, we can create an `AztecOO` object and solve the Schur complement system using our `Schur_Epetra_Operator` object to provide both the Apply() and ApplyInverse() functions. `HardSphereLinProbMgr` reimplements all the linear solver functions in Table 4 to accomplish this.

Finally, the `HardSphereLinProbMgr` class reimplements all functions in Tables 5, 6 and 7. Although not discussed here, it is possible, since these functions are typically called infrequently and not critical to overall application performance, to carefully implement them in `BasicLinProbMgr` and reuse the `BasicLinProbMgr` implementation in `HardSphereLinProbMgr` by providing an auxiliary permutation function that maps GIDs back and forth between the `BasicLinProbMgr` ordering and the `HardSphereLinProbMgr` ordering.

4.4 Packaging the Solver API

One final topic of interest is how to integrate the solver API code into the application. Although it is certainly possible to build the solver API using the application's own build environment, this can require a lot of re-design for the application, especially if it is a Fortran application such as EMU. Instead we have found that using the Trilinos template package (called NewPackage) is an excellent vehicle for encapsulating the solver API code, making it compatible with the Trilinos build system but still independent of Trilinos. Trilinos supports the dynamic registration of independent packages with the Trilinos build process, so the solver API can be built as part of building Trilinos or independently. This approach minimizes the number of required changes to the application build environment.

5   Conclusions

Scientific and engineering applications often require simultaneous solution of multiple degrees of freedom. At the same time, an increasing number of sophisticated preconditioners require a "segregated" view of the linear system. In this paper we have described how to develop an application-specific, solver-generic interface to facilitate the design and development of sophisticated solvers for multi-DOF applications. In particular, we described a working solver API for the molecular density functional theories code Tramonto. We then presented important features of Trilinos, especially the Epetra, AztecOO and Teuchos packages that support construction and use of parallel distributed linear algebra objects for efficient segregated solvers, especially Schur complement approaches. Next, we introduced the `Schur_Epetra_Operator` class that builds on top of the `Epetra_Operator` class and provides an efficient parallel distributed memory Schur complement solver for hard sphere problems. Finally, we sketched out how the `HardSphereLinProbMgr` can be implemented using the `BasicLinProbMgr` interface and the tools in Trilinos.

The general approach discussed here, although specific to Tramonto, should illustrate a potential approach for fully-coupled computational mechanics application, tightly-coupled multi-scale applications and other problem areas where tight interaction exists between disparate degrees of freedom. In particular, this approach opens up the possibility of using the increasingly robust preconditioners from multi-level algorithms research and other low complexity "fast" solvers that, although not appropriate for the fully coupled systems of equations presented by these problem areas, can be applied in combination to subproblems. Work in this area includes, for example, that of Silvester et al. (1999), Kay et al. (2002) and Elman et al. (2003) in CFD.

We expect that, if application developers adopt solver APIs similar to those mentioned here, advances in solver algorithm can be more easily integrated into an application. Furthermore, solver developers will be more able to identify the needs of applications.

REFERENCES

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J. (2003). MUMPS home page. http://www.enseeiht.fr/lima/apo/MUMPS.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. (1995). *LAPACK Users' Guide*. SIAM Pub., Philadelphia, PA, second edition.

Balay, S., Gropp, W., McInnes, L., and Smith, B. (1997). Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M., and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press.

Balay, S., Gropp, W., McInnes, L., and Smith, B. (1998a). PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory.

Balay, S., Gropp, W., McInnes, L., and Smith, B. (1998b). PETSc home page. http://www.mcs.anl.gov/petsc.

Boost.org (2006). Boost C++ libraries. http://www.boost.org/.

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

Dongarra, J. and Eijkhout, V. (2004). Freely available software for linear algebra on the web. http://www.netlib.org/utk/people/JackDongarra/la-sw.html.

Driesen, K. and Holzle, U. (1996). The direct cost of virtual function calls in C++. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–323, New York, NY, USA. ACM Press.

Elman, H., Howle, V. E., Shadid, J. N., and Tuminaro, R. S. (2003). A parallel block multi-level preconditioner for the 3d incompressible navier-stokes equations. *Journal of Computational Physics*, 187(2):504–523.

Falgout, R. (2006). hypre home page. http://www.llnl.gov/CASC/linear_solvers.

Frink, L. J. (2006). Tramonto home page. http://software.sandia.gov/tramonto.

Frink, L. J. D. and Salinger, A. G. (2000a). Two- and three-dimensional nonlocal density functional theory for inhomogeneous fluids i. algorithms and parallelization. *J. Chem. Phys.*, 159:407–424.

Frink, L. J. D. and Salinger, A. G. (2000b). Two- and three-dimensional nonlocal density functional theory for inhomogeneous fluids ii. solvated polymers as a benchmark problem. *J. Chem. Phys.*, 159:425–439.

Heroux, M. A. (2004a). Aztecoo home page. http://software.sandia.gov/Trilinos/packages/aztecoo.

Heroux, M. A. (2004b). Epetra home page. http://software.sandia.gov/Trilinos/packages/epetra.

Heroux, M. A. (2004c). Trilinos home page. http://software.sandia.gov/trilinos.

Heroux, M. A. (2005). Epetra Performance Optimization Guide. Technical Report SAND2005-1668, Sandia National Laboratories.

Heroux, M. A., Frink, L. J., and Salinger, A. G. (2006a). A Schur Complement Based Approach to Solving Density Functional Theories for Inhomogeneous Fluids on Parallel Computers. Technical Report SAND2006–2099, Sandia National Laboratories.

Heroux, M. A., Frink, L. J. D., and Salinger, A. G. (2006b). Segregated Schur complement approaches to solving density functional theories for inhomogeneous fluids on parallel computers. *SIAM J. Sci. Comput.* Submitted.

Heroux, M. A. and Sala, M. (2004). Ifpack home page. http://software.sandia.gov/Trilinos/packages/ifpack.

Kay, D., Loghin, D., and Wathen, A. (2002). A preconditioner for the steady-state navier-stokes equations. *SIAM J. Sci. Comput.*

Li, X. and Demmel, J. (2003). SuperLU home page. http://crd.lbl.gov/ xiaoye/SuperLU/.

R. Pozo (2006). Matrix Market. http://math.nist.gov/MatrixMarket/.

Salinger, A. G., Bou-Rabee, N. M., Pawlowski, R. P., Wilkes, E. D., Burroughs, E. A., Lehoucq, R. B., and Romero, L. A. (2001). *LOCA: A Library of Continuation Algorithms - Theroy and Implementation Manual*. Albuquerque, New Mexico 87185. SAND 2002-0396.

Silling, S. A. (2006). Emu home page. http://www.sandia.gov/emu/emu.htm.

Silvester, D., Elman, H., Kay, D., and Wathen, A. (1999). Efficient preconditioning of the linearized Navier-Stokes equations. Technical Report 352, Manchester, England.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1998). *MPI-The Complete Reference, Volume 1, The MPI core*. The MIT Press.

Stroustrup, B. (2000). *The C++ Programming Language.* Addison-Wesley.

The Mathworks (2006). Matlab documentation homepage. http://www.mathworks.com.

Thornquist, H. K., Bartlett, R. A., Long, K. R., Heroux, M. A., and Sala, M. (2004). Teuchos home page. http://software.sandia.gov/Trilinos/packages/teuchos.

Tuminaro, R. S., Heroux, M. A., Hutchinson, S. A., and Shadid, J. N. (1999). *Official Aztec User's Guide, Version 2.1.* Sandia National Laboratories, Albuquerque, NM 87185.

Tuminaro, R. S. and Hu, J. (2004). Ml home page. http://www.cs.sandia.gov/ tuminaro/ML_Description.html.