Scientific Programming 20 (2012) 241–255
DOI 10.3233/SPR-2012-0352
IOS Press

Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems

Eric Bavier [a], Mark Hoemmen [b,*], Sivasankaran Rajamanickam [b] and Heidi Thornquist [b]
[a] *Cray Inc., St. Paul, MN, USA*
[b] *Sandia National Laboratories, Albuquerque, NM, USA*

Abstract. Solvers for large sparse linear systems come in two categories: direct and iterative. Amesos2, a package in the Trilinos software project, provides direct methods, and Belos, another Trilinos package, provides iterative methods. Amesos2 offers a common interface to many different sparse matrix factorization codes, and can handle any implementation of sparse matrices and vectors, via an easy-to-extend C++ traits interface. It can also factor matrices whose entries have arbitrary "Scalar" type, enabling extended-precision and mixed-precision algorithms. Belos includes many different iterative methods for solving large sparse linear systems and least-squares problems. Unlike competing iterative solver libraries, Belos completely decouples the algorithms from the implementations of the underlying linear algebra objects. This lets Belos exploit the latest hardware without changes to the code. Belos favors algorithms that solve higher-level problems, such as multiple simultaneous linear systems and sequences of related linear systems, faster than standard algorithms. The package also supports extended-precision and mixed-precision algorithms. Together, Amesos2 and Belos form a complete suite of sparse linear solvers.

Keywords: Linear solvers, iterative linear solvers, parallel computing

1. Introduction

Amesos2 and Belos are packages in the Trilinos project [31] written in ANSI C++. Together they provide a complete suite of parallel solvers for large sparse linear systems. Amesos2,[1] a direct methods package, leverages the software investment of several third-party sparse matrix factorization codes by offering an easy-to-use, run-time – configurable interface to all of them. It supersedes Trilinos' Amesos package [49,50]. It improves on Amesos by decoupling the interface whenever possible from the linear algebra objects, so that it accepts arbitrary sparse matrix and vector types. Amesos2 also includes its own "type-generic" factorization for matrices whose entries have any type satisfying a minimal "Scalar" interface. This includes real and complex types, as well as extended-precision floating-point types such as double-double, quad-double [33] and ARPREC [4]. This lets users compute highly accurate factorizations of ill-conditioned matrices.

Belos[2] supersedes Trilinos' AztecOO package [30], which in turn wraps the Aztec library of iterative solvers [58]. Unlike Aztec and many similar libraries, Belos completely decouples the numerical algorithms from the underlying linear algebra objects. This decoupling makes Belos algorithms agnostic of data layout in memory, the distribution of data over processors, and invocations of parallel operations such as reductions. As a result, Belos' performance can track today's rapidly evolving computer architectures without effort. Using Trilinos' Tpetra package to implement linear algebra operations, Belos can already exploit hybrid distributed-memory (via MPI) and shared-memory parallelism, using either CPU-based or GPU-based compute nodes.

Belos implements both *application-aware* and *architecture-aware* algorithms. Application awareness means making the algorithm faster by changing the problem, to one that applications want to solve: linear systems with multiple right-hand sides, or sequences of closely related linear systems. Architecture-aware means that the algorithms reflect how computer architectures have changed: rather than floating-point

*Corresponding author: Mark Hoemmen, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1320, USA. E-mail: mhoemme@sandia.gov.

[1] *Amesos* ($\alpha\mu\varepsilon\sigma o\varsigma$) is a Greek word that means "direct".

[2] *Belos* ($\beta\varepsilon\lambda o\varsigma$) is a Greek word that means "arrow", symbolizing "straight" or "linear".

arithmetic operations, the most expensive operations in terms of performance and energy consumption are communication and data movement [38,60]. This, in part, also involves changing the problem: "block" algorithms for solving multiple right-hand sides simultaneously enable use of faster computational kernels that amortize communication and data movement costs. Furthermore, applying mixed-precision algorithms to solve the problem promises equivalent accuracy but better performance, due to reduced memory bandwidth requirements [12].

This paper gives an overview of the capabilities of the Amesos2 and Belos packages. It explains how their modular software architecture makes it easier for their developers to provide application- and architecture-aware algorithms with maximal code reuse. Amesos2 and Belos rely almost entirely on other Trilinos packages or on third-party libraries for implementations of basic distributed linear algebra operations, preconditioners, and sparse factorizations. Thus, we do not include performance data in this paper, but refer readers to the bibliography for details on the performance of the various Trilinos packages and third-party software.

Section 2 of this paper motivates the development and features of Amesos2 and Belos. The following Section 3 outlines some ways in which Amesos2 and Belos can and do cooperate. In Section 4, we summarize the sparse factorization codes which Amesos2 makes available. We describe Amesos2's software architecture in Section 5. Likewise, in Section 6, we summarize new Krylov subspace algorithms implemented in Belos, and in Section 7, we describe Belos' software architecture. Finally, in Section 8, we discuss future work.

2. Motivation

Amesos2 and Belos both implement functionality which partially exists in the Trilinos packages Amesos and AztecOO, respectively. Thus, it is important for us to justify our effort, both in software engineering and performance terms. In this section, we show how these improvements justify one another: better software engineering makes continuing performance improvements easier.

2.1. Independence from the linear algebra library

Many numerical algorithms can operate abstractly on data objects, without needing to know their internal details. This is, in fact, a main attraction of Krylov subspace methods. They do not need to read or modify the entries of the matrix A, preconditioners, or vectors; they only need to apply A or a preconditioner to a vector, compute dot products, and compute weighted sums of vectors. However, many implementations of Krylov subspace methods do not exploit this flexibility. For example, Aztec and its C++ wrapper, AztecOO, allow a user-defined matrix or preconditioner, but impose specific requirements on the representation and layout of entries in the vectors. Similarly, the "reverse communication" interface that was described in [25], only abstracts away the operation of the matrix A or a preconditioner on a vector, but still requires direct access to the vector's elements. This imposition not only limits software flexibility, it limits performance, because it prevents the linear algebra library from optimizing the representation of data and the computation of basic operations. This is especially important in this time of rapidly evolving computer architectures, as exploiting intranode parallelism and managing data placement become increasingly important (see, e.g., [3,38]). Separating linear algebra data representation and computational kernels from the abstract numerical algorithm frees mathematicians from tracking rapid developments in computer architectures, and gives their software a longer useful life. For a more detailed discussion of the value of separating linear algebra objects from abstract numerical algorithms, see [7].

2.2. Maximize code reuse

Popular scientific codes such as LAPACK [2] stay in use for decades, and take generations of highly trained scientific programmers to maintain. This suggests that design practices should favor code reuse, modularity, and generality. Amesos2 and Belos have followed this principle in their modular software architecture, described in Sections 5 and 7, respectively. Learning how to specialize Amesos2's or Belos' C++ traits interface takes much less time than reimplementing numerical algorithms for each linear algebra library. The abstract traits interface lets Amesos2 and Belos developers express algorithms in as mathematical a language as possible, since scientific programmers prefer "describ[ing] the algorithms in a mathematical language as opposed to a computer language" [9]. The use of C++ traits classes with compile-time specializations to access basic linear algebra operations means that the "abstraction penalty" per kernel invocation at run time may be zero (with successful inlining), and is at most

one function call (for the entire kernel). Compile times do increase with C++ templates. However, a few minutes suffice to build and link an entire solver stack (including multiple distributed linear algebra implementations, solvers, and preconditioners) from scratch, with full optimization enabled, including a complete suite of tests. Writing code that works without changes for data of any Scalar type takes a little extra care, but saves effort over maintaining N copies of the code for each of N Scalar types. Finally, the run-time customization capability of Amesos2 and Belos helps programmers avoid programming. Zero lines of code have zero bugs.

2.3. Application- and architecture-aware algorithms

Many users might ask for a routine that solves a single linear system $Ax = b$. However, their application actually calls for solving many linear systems, either:

- with the same matrix, but many right-hand sides at once ($AX = B$, i.e., $A[x_1, \ldots, x_n] = [b_1, \ldots, b_n]$),
- with the same matrix, but many right-hand sides, only available in sequence ($Ax_i = b_i$, $i = 1, 2, \ldots$), or even
- sequences of closely related linear systems ($(A + \Delta A_i)x_i = b_i$, $i = 1, 2, \ldots$).

This is often the case in applications performing parameter studies, propagation of uncertainty in forcing terms, and nonlinear time-dependent analysis [45]. Solving these higher-level problems, instead of just solving one linear system at a time, often results in more efficient algorithms. We call such algorithms *application aware*. Belos implements application-aware algorithms for solving the above problems: *block* and *pseudoblock* iterative methods for $AX = B$, and *recycling* iterative methods for sequences of closely related linear systems $Ax_i = b_i$ or $(A + \Delta A_i)x_i = b_i$. For more details, see Section 6.

Architecture-aware algorithms, in turn, have a design influenced by an understanding of how much different operations cost on modern computer architectures. Data movement and communication between parallel processors is much slower than floating-point arithmetic on modern machines, and also consumes much more energy. Block and pseudoblock solvers are more architecture-aware than standard iterative methods, because they can use more efficient computational kernels that amortize communication costs over multiple vectors. Standard algorithms are stuck with slower kernels, such as SpMV (sparse matrix–vector multiply) and vector–vector operations. Block methods can use kernels like SpMM (sparse matrix times multiple dense vectors) and block vector operations. SpMV has performance dominated by data movement, in particular by reading the entries of the sparse matrix; SpMM amortizes this cost over multiple vectors (see, e.g., [27,36,37,39,42]). The cost of vector–vector operations (vector sums and inner products) is dominated by global parallel reductions and by reading and writing the vector entries. Blocking up the vector operations amortizes the communication cost and enables use of faster BLAS 3 operations (see, e.g., [14,24,34, 54,55]). In the case of block solvers, architecture and application awareness coincide happily.

Another way in which Amesos2 and Belos are architecture aware, is in mixed-precision algorithms. Memory bandwidth is a scarce resource on modern processors. Reading and writing lower-precision floating-point numbers takes less bandwidth, but can sacrifice accuracy. *Mixed-precision* algorithms can regain much of this accuracy, by using more precise floating-point types to improve the result of lower-precision computations [12]. Amesos2 and Belos support mixed-precision computation natively. Both packages' interfaces accept matrices and vectors with entries of arbitrary "Scalar" type. Furthermore, they allow linear algebra objects and solvers with different Scalar types to coexist in the same program. Many solver libraries are written "abstractly" on the data type, but that abstraction is implemented via a C typedef rather than by a C++ template parameter. Using a typedef means that Scalar is fixed to a single type when the library is built. Amesos2 and Belos have no such restriction. This enables mixing objects and algorithms of different floating-point precisions in the same execution unit. This can be exploited for the development of novel "adaptive-precision" algorithms. We describe an example of this in Section 7.5.

2.4. Package-specific motivations

Amesos2 inherits the same motivations as Amesos, namely to provide an interface that makes it easy to call any of several sparse direct factorization codes [50]. No one direct linear solver is best overall, even for problems in the same class. For example, see [21] for one comparison between supernodal and non-supernodal direct solvers. Each direct solver has many configuration options, which are represented in incompatible ways. Most importantly, sparse factorizations

are some of the most complicated codes to implement because of their intrusive manipulation of sparse matrix structure. A reasonable description of these manipulations, even with minimal optimization, takes most of a recently published book [18]. Thus, it is better to leverage that software investment, rather than reimplement sparse factorizations with the desired interface and features. Nevertheless, sparse factorizations have the potential for abstractness, since they interact with the structure and entries of the sparse matrix in limited ways. Amesos2 exploits this via its C++ traits interface that allows the library to accept any sparse matrix data type, with high-performance specializations for certain types.

Belos inherits many of the same motivations as AztecOO, in that it provides a suite of iterative linear solvers. However, Belos distinguishes itself from AztecOO in that these solvers are implemented in a framework that promotes interoperability, extensibility and reusability. A basic implementation of any Krylov subspace method rarely takes more than half a page to describe (see, e.g., [6]). A more abstract view of any Krylov subspace method can break down this implementation into several components: subspace generation (iteration), orthogonalization, stopping criteria (status testing), and the linear problem. An appropriately designed framework enables a user or developer to vary any of these components with little or no need to rewrite an entire solver. Belos' framework provides these algorithmic components (see Section 7.4) and a C++ traits interface for linear algebra (see Section 2.1) to address the needs of today's user and adapt to the needs of tomorrow's user.

3. Cooperation of Amesos2 and Belos

The most common way in which sparse direct and iterative solvers cooperate involves using the former to construct preconditioners for the latter. Amesos2's complete and incomplete factorizations can be used directly as preconditioners, as the block solver in a domain decomposition preconditioner, such as block Jacobi, or as smoothers within a multilevel algebraic preconditioner. Plugging any preconditioner into Belos requires minimal effort to implement the traits interface. Amesos2 and Belos can also cooperate in novel ways, for example in the new "hybrid" direct-iterative Schur complement-based block solver ShyLU [46]. Finally, Amesos2 and Belos can use the same linear algebra objects (sparse matrices and vectors). This makes it easy to compare direct and iterative solvers, or even to use a direct solver as a "backup method" for robustness.

4. Algorithms provided by Amesos2

Amesos2's users have different application-specific use cases for direct solvers. The three common use cases come from parallel scalability considerations. First, users with large ill-conditioned matrices require distributed-memory parallel direct solvers in order to solve the problem accurately. Second, hybrid direct-iterative solvers like ShyLU [46] require a shared-memory parallel direct solver. Finally, smoothers within a multilevel preconditioner require a sequential direct solver. In order to meet the needs of all users, Amesos2 supports all three direct solvers from the SuperLU family: sequential SuperLU [22], multithreaded SuperLU-MT [23] and distributed-memory parallel SuperLU-Dist [40]. We also provide an interface to the multithreaded direct solver PARDISO [51]. In addition, Amesos2 includes the direct solver KLU [21]. KLU is particularly effective for matrices on which supernodal factorizations do not perform well, such as matrices from circuit simulations. It also serves as a fall-back in case no external solvers are available.

Amesos2 provides access to mixed-precision algorithms in two ways. First, its native KLU solver is templated on the type of matrix entries, so it can factor and solve sparse linear systems whose entries have any Scalar type. Second, Amesos2's interface exposes the Scalar types supported by each third-party solver library, by mapping the user's data type to a compatible type of equal or greater precision which the library supports. Amesos2 allows users to access solvers for different precisions at the same time, without requiring recompilation.

5. Amesos2 software architecture

5.1. *Amesos*2 design assumptions

Amesos2's primary design goal is to be a single interface for multiple third party direct solvers. Amesos2 assumes that the direct solvers implement four operations: preordering, symbolic factorization, numeric factorization and triangular solve. While this does not require that the direct solver provide a separate interface to all four stages, Amesos2 may exploit a four-stage interface to improve performance. For example, while some direct solvers combine the preordering and the symbolic factorization into one phase, Amesos2 can replace the solver's native preordering with a more efficient method. For example, Zoltan's hypergraph partitioning [11] is useful as a fill-reducing ordering for unsymmetric matrices. When the direct solvers have

the preordering as an option, it may be more efficient to use our ordering methods and skip the native ordering of the solvers.

Amesos2 is designed to be used with different types of sparse matrices and dense vectors. Amesos2 assumes that the matrices and vectors are heavyweight objects, which the user passes in either by pointer or by reference-counted pointers from the Teuchos memory management classes [8]. Depending on the third-party direct solver being used, Amesos2 might copy the matrix and vector once into the format required by the direct solver. It holds the symbolic and numeric factorization and the internal data structures of the direct solver, until the input matrix itself is reset to a new matrix or the Amesos2 Solver object itself is destroyed.

5.2. *Typical Amesos2 usage*

The flowchart in Fig. 1 summarizes several different use cases for Amesos2 solvers. Amesos2 supports many different use cases, in part because it is designed with both novice and expert users in mind. Novice users usually have one relatively small linear system and want to solve it with a direct linear solver. This simple case can be achieved with two lines of code in Amesos2: Create a solver instance of specific type with the matrix, left-hand side (solution) vector and the right-hand side vector, then call the `solve()` method on the solver instance. Amesos2 does the local ordering and the symbolic and numeric factorization if needed.

Assuming that a sparse matrix A of type `MAT` and input and output vectors X, respectively, B of type `MV` already exist, here is code illustrating the simplest solve case, using SuperLU as the underlying solver.

```
Teuchos::RCP<Amesos2::Solver<MAT,MV>> solver=
  Amesos2::create<MAT,MV> ("Superlu",A,X,B);
solver->solve ();
```

`RCP`, one of the Teuchos Memory Management classes, represents a reference-counted "smart" pointer. Both Amesos2 and Belos depend heavily on these classes to manage shared ownership of heavyweight data safely and efficiently. From now on, all examples will begin with the following code:

```
using Teuchos::RCP;

RCP<Amesos2::Solver<MAT,MV> > solver =
  Amesos2::create<MAT,MV> ("Superlu",A,X,B);
```

More sophisticated users can separate the preordering and symbolic and numeric factorization steps. After each step, the solver can report useful information, like the total number of stored (structurally nonzero) entries in the L and U factors. This is useful for computing memory usage and comparing the effectiveness of different preorderings and solver algorithms.

```
solver->numericFactorization ();
Amesos2::Status solver_status =
  solver->getStatus ();
std::cout
  << "Number of entries stored in L+U: "
  << solver_status.getNnzLU()
  << std::endl;
solver->solve ();
```

Expert users of Amesos2 require finer-grained control of local ordering, symbolic and numeric factorization. They will often compute the preordering and symbolic factorization once for a sequence of matrices with the same nonzero pattern. When the matrix values have changed, the numeric factorization is then computed and used, one or more times, to solve the linear problem. Almost all direct solvers support this use case without any changes. Amesos2 supports this as well; its interface lets users change the matrix and specify that the next call to `solve()` will perform the numeric factorization on the new matrix using the existing preordering and symbolic factorization. Furthermore, individual steps of a direct solve can be called explicitly for this new matrix. Most direct solvers support using the same solver instance for solving multiple linear systems. For examples, see the Amesos2 source directory.

Amesos2 accepts two optional sets of parameters. The first set of parameters controls Amesos2 itself and supports the most common options among the direct solvers. The second set of parameters correspond to the specific solver to use. While it is not a simple task to support every option supported by every direct solver and maintain that across multiple versions of the solvers, the infrastructure is in place to do that. The list of parameters will be maintained based on the needs of Amesos2 users. Here is an example of a simple solve with SuperLU that sets parameters:

```
using Teuchos::ParameterList;
using Teuchos::parameterList;

// Create a ParameterList to hold solver
// parameters
RCP<ParameterList> amesos2Params =
  parameterList ("Amesos2");
```

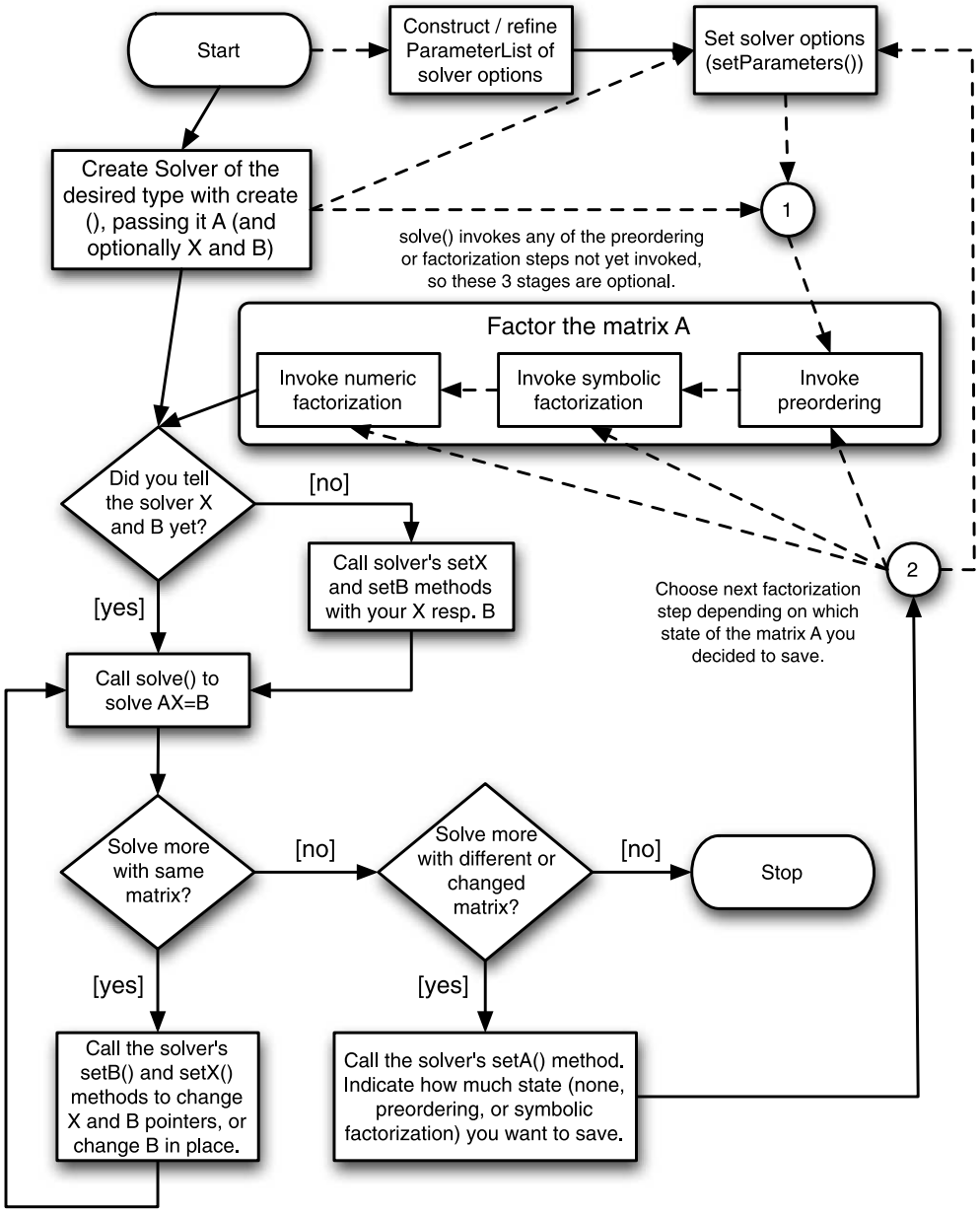

Fig. 1. Flowchart of different Amesos2 use cases for solving linear systems. Dotted connectors denote optional steps. We write $AX = B$, rather than $Ax = b$, to show that each "solve" invocation may be applied to multiple right-hand sides.

```
ParameterList& superluParams =
  amesos2Params->sublist ("SuperLU");
superluParams.set ("Trans", "TRANS");
// Solve with A^T
// Don't equilibrate the system before solve.
superluParams.set ("Equil", false);
// Use the "natural" column ordering.
superluParams.set ("ColPerm", "NATURAL");
solver->setParameters (amesos2Params);
solver->solve ();
```

5.3. Interface to matrix and vector types

Amesos2 maximizes code reuse with compile-time polymorphism. One way in which Amesos2 does so is in its support for various sparse matrix and dense vector types. Amesos2's Solver interface has two template parameters: Matrix and Vector. The former represents a sparse matrix, and the latter a collection of one or more dense vectors, which are the right-hand

side(s) and solution vector(s). Amesos2 accepts different matrix and vector types via compile-time specializations of adapters: MatrixAdapter for Matrix, and MultiVecAdapter for Vector. Solvers in turn use these adapters to access the data, either directly if the solver supports Matrix and Vector's native data structures, or by copy otherwise.

One important motivation for Amesos2's compile-time polymorphic architecture is to support linear systems with more than two billion unknowns, as that was a significant limitation of Amesos. This requires support for integer indices larger than 32 bits. Another motivation for Amesos2 is support for mixed-precision computations. Trilinos' Tpetra package of distributed linear algebra objects helps Amesos2 achieve both goals. Tpetra templates its matrix and vector classes on both the "Scalar" (type of matrix or vector entries) and "Ordinal" (integer index) types. Tpetra objects may have several template parameters, but Amesos2 hides this complexity by templating its Solver interface on only the Matrix and Vector types.

This feature also simplifies adding support for new matrix and vector types. For example, a PETSc sparse matrix [5] can be supported just by specializing the MatrixAdapter template class. The matrix adapter only requires methods to access a compressed row (or column), matrix attributes like the dimensions, a method for describing the matrix's distribution over distributed-memory processes, and an "import" method to redistribute the matrix if the solver requires it. Once this adapter is written all solvers supported by Amesos2 will work with the adapter. Amesos2 currently includes adapters for the following Trilinos sparse matrices:

- `Epetra_CrsMatrix` and `Epetra_RowMatrix`,
- `Tpetra::CrsMatrix` and `Tpetra::RowMatrix`

and the following Trilinos dense vectors:

- `Epetra_MultiVector`,
- `Tpetra::MultiVector`.

5.4. Interface to solvers

Adding an adapter for a new solver in Amesos2 takes about as much effort as adding a new matrix or vector adapter. The third party solver's interface must at least separate the numeric factorization and solve into two steps. Even LAPACK's dense LU factorization does this with the GETRF, respectively, GETRS routines, and Amesos2 thus even offers an LAPACK interface. (It first makes the sparse matrix dense, which might be a reasonable choice for a sufficiently small matrix.) However, Amesos2 can expose more optimizations if the third party solver provides a separate interface for all four steps mentioned above. In addition, the third-party solver must have a way to get and set the solver-specific parameters and to check if the matrix type is compatible with the solver. Amesos2 Solver objects must be templated on the matrix and vector types and use the matrix and vector adapters to convert the matrix and vector objects to the data structures required by the third-party direct solver. As a result, the latter will support all the matrices and vectors Amesos2 supports without any specializations (provided the third party solver is able to handle all the "Scalar" and "Ordinal" types).

6. Algorithms implemented in Belos

Belos provides implementations of several Krylov subspace methods, listed in Table 1 along with the C++ class that exposes each method to users. For symmetric positive definite systems, Belos implements several variants of CG (the Method of Conjugate Gradients of Hestenes and Stiefel [32]). For symmetric indefinite linear systems, Belos offers MINRES, the Minimum Residual method of Paige and Saunders [43]. Belos includes many variants of GMRES (the Generalized Minimal Residual method of Saad and Schultz [48]), Flexible GMRES [47] and a Transpose-Free Quasi-Minimal Residual (TFQMR, of Freund [28]) implementation. Finally, Belos has an implementation of Paige and Saunders' LSQR iteration [44] for solving linear and damped least-squares problems, as well as possibly singular nonsymmetric linear systems.

Belos does not claim to offer a complete suite of Krylov subspace methods. Developers have focused on methods most trusted by domain experts for their robustness, in particular on variants of GMRES. Belos particularly has only two short-recurrence methods for nonsymmetric linear systems – TFQMR and LSQR – despite the large number of such methods available in the literature. This is because users consider them less robust than GMRES. The decreasing amount of memory expected per node on future large-scale parallel computers (see, e.g., [38]) may make nonsymmetric short-recurrence solvers more attractive in the future. The Belos solver framework enables rapid devel-

Table 1
List of Krylov subspace methods implemented in Belos, along with the type of algorithm and the SolverManager subclass which provides the method

Algorithm	Type of solver	SolverManager subclass
CG	Single RHS	BlockCGSolMgr
Block CG	Block	BlockCGSolMgr
Pseudoblock CG	Pseudoblock	PseudoBlockCGSolMgr
Recycling CG	Recycling	RCGSolMgr
PCPG	Seed	PCPGSolMgr
Block GMRES	Block	BlockGmresSolMgr
Block FGMRES	Block	BlockGmresSolMgr
Pseudoblock GMRES	Pseudoblock	PseudoBlockGMRESSolMgr
Recycling GMRES (GCRO-DR)	Recycling	GCRODRSolMgr
Hybrid Block GMRES	Seed	GmresPolySolMgr
MINRES	Single RHS	MinresSolMgr
Transpose-Free QMR (TFQMR)	Single RHS	TFQMRSolMgr
LSQR (least squares)	Single RHS	LSQRSolMgr

Note: "Single RHS" means that the algorithm can only solve for one right-hand side at a time.

opment of any desired iterative linear solver, so adapting to application- and architecture-focused needs is facilitated by its design.

6.1. Block vs. pseudoblock solvers

Block iterations are mathematically different algorithms from their single-vector counterparts. This means they have different algorithmic performance characteristics, such as the number of iterations to meet the same convergence criteria. Many Belos users want the computational performance benefit of block iterative methods, with the same convergence behavior as single-vector methods. Belos provides this with its "pseudoblock" solvers. These execute the single-vector algorithm for each right-hand side in "lock step," by applying the matrix A and any preconditioners to all vectors in a block at once and using block vector operations. If one or more of the linear systems meet the convergence criteria before the rest, the solver "deflates" them by constructing a view of the unconverged right-hand sides, and continuing the iteration on those. Belos provides two pseudoblock solvers: Pseudoblock GMRES and Pseudoblock CG.

6.2. Recycling solvers

Krylov subspace recycling attempts to accelerate the convergence for a sequence of linear systems

$$(A + \Delta A_i)x_i = b_i, \quad i = 1, 2, \ldots,$$

through the judicious selection and use of a projection subspace between one solve and the next [45]. This technique has proven effective for sequences of closely related linear systems, like those found in modeling fatigue and fracture via finite element analysis. Recycling is also effective when performing restarting within one linear system [41,56]. Belos provides two single-vector recycling solvers: Recycling GMRES (GCRO-DR) and Recycling CG. Sandia National Laboratories and Temple University are currently collaborating on a Block Recycling GMRES algorithm to be deployed in Belos [53,61].

6.3. "Seed" solvers

Krylov subspace methods that attempt to accelerate the convergence for a sequence of linear systems

$$Ax_i = b_i, \quad i = 1, 2, \ldots,$$

where the right-hand sides are not available all at once, are called "seed" solvers. These solvers either use a random vector or b_1 to create a subspace or polynomial filter to be applied during each solve to accelerate convergence. This subspace and polynomial can be updated from one solve to the next. Recycling solvers can also be considered seed solvers, as they could easily be applied to sequences of linear systems where the matrix does not change. However, Belos provides two seed solvers specifically for this use case: Hybrid Block GMRES and PCPG.

6.4. Least-squares solvers

Least-squares solvers can always solve $Ax = b$ in a least-squares sense, even if the matrix A is singular

or the system is inconsistent. Belos provides the least-squares solver LSQR [44], which computes a monotonically increasing lower bound of the condition number of A, defined as $\|A\|\|A^\dagger\|$, where $A^\dagger$ is the pseudoinverse of A. This solver is specifically useful for experimentation with mixed-precision algorithms, which will be discussed in Section 7.5.

7. Belos software architecture

7.1. Belos design assumptions

Belos' design goal is to offer a generic interface to a collection of iterative methods for solving large, sparse linear systems. For algorithm developers, Belos provides algorithmic components that facilitate extensibility and reusability with the express intent of simplifying the implementation of complex algorithms. Incorporated into the Belos design is the assumption that iterative methods can be decomposed into several components, including: orthogonalization (`OrthoManager`), stopping criteria (`StatusTest`), subspace construction (`Iteration`), and a solution strategy (`SolverManager`). The linear problem, itself, is described by a separate class (`LinearProblem`) that incorporates necessary preconditioning or scaling of the linear problem, as defined by the user. These algorithmic components will be discussed in Section 7.4.

Belos also makes several assumptions regarding the underlying linear algebra objects. These have been incorporated into the design of the operator and vector traits interfaces. Similar to Amesos2, Belos assumes that the operators and vectors are heavyweight objects, meaning that sparse matrices and preconditioners are time-consuming to compute and memory for copies of vectors is limited. This is handled with minimally error-prone explicit memory management, first by using Trilinos' Teuchos memory management classes [8] as handles for heavyweight objects, and second by supporting both read-only and read-write views of vectors.

Belos' operator and vector traits interface has been simplified through some parallelism assumptions. First, the concrete linear algebra objects are expected to handle all explicit communication, so that the implementation of any algorithm looks as much like mathematics as possible. Furthermore, the result of any reduction operation (dot product, norm, etc.) on a vector or set of vectors (multivector) is expected to be replicated over all participating distributed-memory processes. This means that Belos only exploits intranode shared-memory parallelism if the concrete linear algebra objects do. Belos does not introduce its own shared-memory parallelism for the small dense linear algebra operations required by many Krylov methods. Furthermore, rounding errors in reduction operations may result in situations where different processes take divergent paths through the solver. Heterogeneous nodes (see [10]) or the use of nondeterministic shared-memory parallelism may exacerbate this problem. This has not yet proven an issue in practice.

7.2. Typical Belos usage

The flowchart in Fig. 2 shows the standard use case for Belos solvers. A Belos `SolverManager` requires two items for construction: a `LinearProblem` object and a set of options that are stored and passed in using a `ParameterList`. The latter, a class in the Teuchos Trilinos package, maps option names to option values. It allows hierarchical nesting, where an option's value may itself be a `ParameterList`. A user can generate this parameter list in two ways: construct a minimal list containing a subset of options with non-default values, or acquire the solver's default options by calling `getValidParameters()` on the solver object and then modify that list as desired. The `LinearProblem` object contains the matrix A, the right-hand side(s) B, the initial guess(es) X, and a left, right or both (split) preconditioner(s). The loop in the flowchart returning to circle "A" shows that the same `SolverManager` object and parameters can solve multiple linear systems in sequence. This avoids expensive reconfiguration and rebuilding of state. In addition, some Belos solvers, such as the recycling (Section 6.2) and seed (Section 6.3) solvers, may save state computed from the first solve in order to accelerate subsequent solves. This behavior happens without user intervention, though users can invoke the solver's `reset()` method to clear out this state for recomputation by the next `solve()` call.

Here follows a simple example of how to use GMRES to solve a given linear system $AX = B$ with a right preconditioner M. The matrix A and preconditioner M have type `OP`, representing an operator such as `Epetra_Operator` or `Tpetra::Operator`, and the vectors have type `MV`, representing a vector such as `Epetra_MultiVector` or `Tpetra::MultiVector`. We simplify the example by using a "factory" to create the solver, though you can also create specific solvers directly by invoking their constructors (Listing 1).

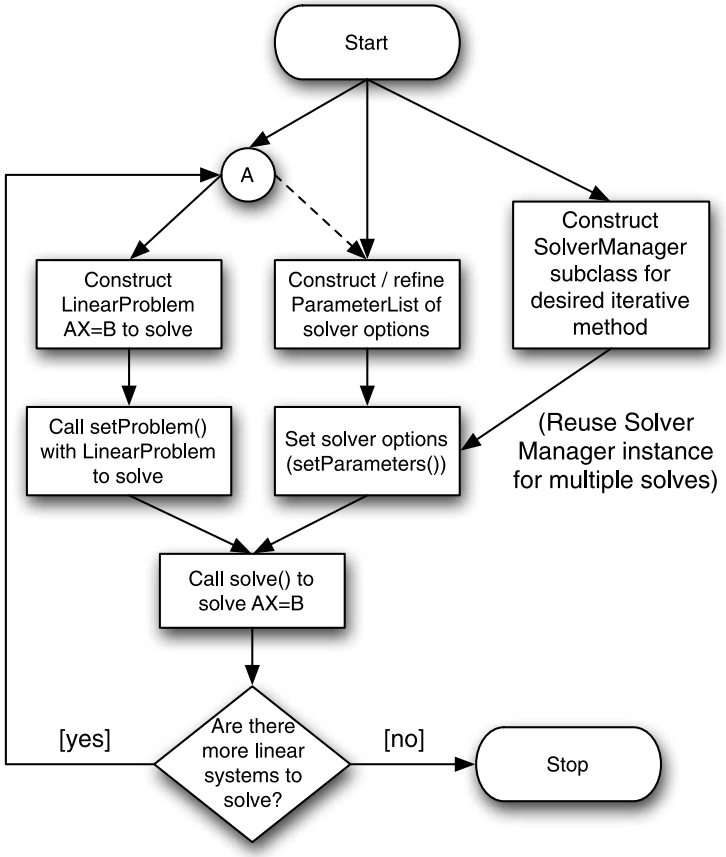

Fig. 2. Flowchart of typical Belos usage for solving linear systems. Dotted connectors denote optional steps. We write $AX = B$, rather than $Ax = b$, to show that each "solve" invocation may be applied to multiple right-hand sides.

7.3. *Interface to matrices, vectors and preconditioners*

The Belos solver framework uses C++ templates for compile-time polymorphism. All of the algorithmic components have three template parameters: a scalar type (Scalar), a multivector type (MV), and an operator type (OP). The scalar type identifies the type of entries in the multivector and operator. The multivector type is used to represent the right-hand side(s) (B) and the solution vector(s) (X). The operator type is used to represent the matrix A and any preconditioner(s). Each operator is expected to interact with the multivector type MV by taking a MV reference as input, and writing the result of applying the operator (or its transpose or conjugate transpose, if supported) to another MV reference.

Rather than requiring the specific scalar, multivector, and operator types to support operations directly, Belos uses a C++ traits interface to implement compile-time polymorphism. The scalar traits are provided by the `ScalarTraits` class in the Teuchos package of Trilinos, and the multivector and operator traits are provided through Belos' `MultiVecTraits` and `OperatorTraits` class, respectively. Belos currently provides implementations of `MultiVecTraits` and `OperatorTraits` for linear algebra objects from three different Trilinos packages: Epetra, Tpetra, and Thyra. In addition, users may specialize the traits interface themselves. Finally, a run-time polymorphic interface is available through Belos' own `MultiVec` and `Operator` abstract interfaces.

7.4. *Belos algorithmic components*

The Belos design enables the implementation of any iterative method using algorithmic components, including: the linear problem (`LinearProblem`), orthogonalization (`OrthoManager`), stopping criteria (`StatusTest`), subspace construction (`Itera-`

```
using Teuchos::ParameterList;
using Teuchos::parameterList;
using Teuchos::RCP;
using Teuchos::rcp; // Save some typing

// The ellipses represent the code you would normally use to
// create the sparse matrix, preconditioner, right-hand side,
// and initial guess for the linear system AX=B to solve.
RCP<OP> A = ...; // The sparse matrix / operator A
RCP<OP> M = ...; // The (right) preconditioner M
RCP<MV> B = ...; // Right-hand side of AX=B
RCP<MV> X = ...; // Initial guess for the solution

// Make an empty new parameter list.
RCP<ParameterList> solverParams = parameterList();

// Set some GMRES parameters.
//
// "Num Blocks" = Maximum number of Krylov vectors to store.
// This is also the restart length. "Block" here refers to
// the ability of this particular solver (and many other Belos
// solvers) to solve multiple linear systems at a time, even
// though we are only solving one linear system in this example.
solverParams->set ("Num Blocks", 40);
solverParams->set ("Maximum Iterations", 400);
solverParams->set ("Convergence Tolerance", 1.0e-8);

// Create the GMRES solver using a "factory" and
// the list of solver parameters created above.
Belos::SolverFactory<Scalar, MV, OP> factory;
RCP<Belos::SolverManager<Scalar, MV, OP> > solver =
  factory.create ("GMRES", solverParams);

// Create a LinearProblem struct with the problem to solve.
// A, X, B, and M are passed by (smart) pointer, not copied.
RCP<Belos::LinearProblem<Scalar, MV, OP> > problem =
  rcp (new Belos::LinearProblem<Scalar, MV, OP> (A, X, B));
problem->setRightPrec (M);

// Tell the solver what problem you want to solve.
solver->setProblem (problem);

// Attempt to solve the linear system. result == Belos::Converged
// means that it was solved to the desired tolerance. This call
// overwrites X with the computed approximate solution.
Belos::ReturnType result = solver->solve();

// Ask the solver how many iterations the last solve() took.
const int numIters = solver->getNumIters();
```

Listing 1. A simple example to use GMRES to solve a given linear system $AX = B$ with a right preconditioner M.

tion) and a solution strategy (`SolverManager`). We briefly summarize each of these essential components here.

7.4.1. Linear problem

A `LinearProblem` object is a container for operator A, the right-hand side(s) B, the initial guess(es) X, and a left, right or both (split) preconditioner(s). This class defines a minimum interface that can be expected of all linear problems by the classes that will work with these problems. The methods provided by this interface are generic enough to define any linear problem that is Hermitian or non-Hermitian. The

`LinearProblem` class provides a default implementation of these methods, but a user can modify this using run-time polymorphism.

7.4.2. Orthogonalization

Orthogonalization and orthonormalization are commonly performed computations in iterative linear solvers and can be implemented in a variety of ways. The `OrthoManager` class separates the `Iteration` from this functionality. The `OrthoManager` defines a small number of orthogonalization-related operations, including a choice of an inner product. The OrthoManager interface has also been extended, through inheritance, to support orthogonalization and orthonormalization using matrix-based inner products in the `MatOrthoManager` class.

Belos provides several different orthogonalizations, which offer trade-offs between accuracy and performance. Users can experiment with different orthogonalization methods and their parameters by setting run-time options in the solver parameter list. The orthogonalization methods provided by Belos work on blocks of vectors, and are valid for both Euclidean [55] and non-Euclidean [57] inner products. Belos supports this in two ways: by passing an inner product operator B (such that $\langle \cdot, \cdot \rangle_B$, and B is Hermitian positive definite) to a `MatOrthoManager` and by changing the inner product method (`MvTransMv`) in the interface to the multivector.

Belos provides four concrete orthogonalization managers:

- `DGKSOrthoManager` – performs "Classical Gram–Schmidt (CGS)" with a DGKS correction [15];
- `ICGSOrthoManager` – performs "Iterated Classical Gram–Schmidt" (ICGS);
- `IMGSOrthoManager` – performs "Iterated Modified Gram–Schmidt" (IMGS);
- `TsqrMatOrthoManager` – performs "Tall Skinny QR (TSQR)" as the normalization step [35].

TSQR provides better performance and accuracy than MGS or CGS when normalizing blocks with multiple columns. However, TSQR is currently only available for the four Scalar types supported by LAPACK (real and complex IEEE 754 single- and double-precision floating-point values). Furthermore, TSQR currently only supports orthogonalization with respect to the Euclidean inner product, though algorithms exist for the general inner product case and can be implemented given sufficient interest. For solvers that support different orthogonalization methods, users may select the method and its parameters (such as reorthogonalization thresholds) via the `ParameterList` for the solver of their choice.

7.4.3. Stopping criteria

Belos provides a generic interface called `StatusTest` for stopping criteria. Solvers construct implementations of this interface to control termination of the subspace construction (`Iteration`). Users can also provide custom stopping criteria by implementing their own `StatusTest` subclass and passing an instance of it to the solver. Belos provides classes for composing `StatusTest` instances, so that the resulting composite test passes if Boolean combinations of the constituent tests passed (optionally with short-circuiting semantics to avoid unnecessary test evaluations). A `StatusTest` instance "passes" when it thinks the iteration should stop. This may indicate positive results (e.g., the method has converged to the desired relative residual tolerance) or negative results (e.g., the maximum number of iterations has been reached). The solution strategy, implemented by the `SolverManager`, must determine the reason for the termination of the `Iteration` by interpreting the results of the various stopping criteria, and must know how to proceed. Belos' design assumes that the `StatusTest` is evaluated redundantly over all distributed-memory processes and that the returned Boolean value is the same on all processes. However, if inconsistent parallel convergence tests are necessary, Belos' modular design makes it easy to change convergence tests to require agreement between all processors before terminating the iteration.

7.4.4. Subspace construction

The `Iteration` class provides generic computational kernels that do not have the intelligence to determine when to stop the iteration, what the linear problem of interest is, or how to orthogonalize the basis for a subspace. The intelligence to perform these three tasks is, instead, provided by the `StatusTest`, `LinearProblem` and `OrthoManager` objects, which are passed into the constructor of an `Iteration`. This allows each of these three tasks to be modified without affecting the basic solver iteration. When combined with the status and state-specific methods provided by the `Iteration` class, this gives the user a large degree of control over linear solver iterations.

7.5. Enabling mixed-precision algorithms

Solving problems with the least precision possible saves both storage and memory bandwidth, which are scarce resources on modern computers and likely to become scarcer. However, an ill-conditioned matrix A might be numerically rank-deficient at lower precisions. This may affect accuracy of computed preconditioners and certain linear solvers. LSQR's exact condition number lower bound enables a new kind of algorithm: one which dynamically increases floating-point precision until it knows it can solve the problem accurately. LSQR can detect and correct this problem by first attempting to solve the linear system at the lowest precision possible (e.g., IEEE 754 single precision). If it finds that the condition number of A is greater than the inverse of machine precision at the current working precision, then A is numerically rank deficient at working precision. An outer loop around LSQR can then increase working precision and solve again, increasing precision until it finds one at which A is numerically full rank. The user can then confidently use that precision for successive solves with A, using either LSQR or another method.

8. Future work

8.1. Amesos2 plans

Amesos2 plans to continue expanding software support for different matrix and vector representations found within Trilinos. We also may add support for matrix representations from other software packages such as PETSc [5] or Hypre [26] if the need arises. Most direct solvers support a simple compressed row or column format. We plan to support this data structure as well. Amesos2 also needs an interface to more direct solvers, such as PaStiX [29], MUMPS [1] and UMFPACK [16,17,19,20]. Finally, we plan to support Cholesky factorization with an interface to CHOLMOD [13]. This improves upon the previous package Amesos, which lacked a Cholesky factorization capability.

8.2. Belos plans

Belos also offers a modular framework for research and development of new, as well as known, iterative solvers. It currently provides only two short-recurrence iterative methods for nonsymmetric linear systems, namely Transpose-Free QMR (TFQMR) [28] and LSQR [44], as opposed to many GMRES variants. This is because application developers are more familiar with GMRES' robustness, but longer vector recurrence means that it takes more memory. The decreasing amount of memory expected per node on future large-scale parallel computers (see, e.g., [38]) may make short-recurrence methods for nonsymmetric systems, like BiCGSTAB [59] or IDR [52], attractive in the future.

Implementation of communication-avoiding Krylov subspace methods [34] is ongoing in Belos. These algorithms replace the kernels in standard Krylov methods with new kernels that communicate less between processors and move less data between levels of the memory hierarchy. The Tall Skinny QR (TSQR) factorization is one of these kernels, and is currently available as a `MatOrthoManager` [35]. Implementation of another component, the "matrix powers kernel" that computes a Krylov subspace basis with minimal communication, is ongoing work involving a collaboration between Sandia National Laboratories and the University of California Berkeley.

Acknowledgements

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] P. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, B. Ucar, F.-H. Rouet, C. Weisbecker, M.W. Sid-Lakhdar, G. Joslin and M. Brémond, MUMPS webpage, http://mumps.enseeiht.fr/ (last accessed 16 April 2012).

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, 1999.

[3] K. Asanovic, R. Bodik, J.W. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D.A. Patterson, K. Sen, J. Wawrzynek, D. Wessel and K.A. Yelick, A view of the parallel computing landscape, *Commun. ACM* **52** (2009), 56–67.

[4] D.H. Bailey, Y. Hida, X.S. Li and B. Thompson, ARPREC: an arbitrary precision computation package, Technical Report LBNL-53651, Lawrence Berkeley National Laboratory, September 2002, available at: http://crd.lbl.gov/~dhbailey/dhbpapers/ (last accessed 17 April 2012).

[5] S. Balay, J. Brown, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith and H. Zhang, PETSc webpage, http://www.mcs.anl.gov/petsc (last accessed 17 April 2012).

[6] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H.V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd edn, SIAM, Philadelphia, PA, 1994.

[7] R.A. Bartlett, Thyra linear operators and vectors: overview of interfaces and support software for the development and interoperability of abstract numerical algorithms, Technical Report SAND2007-5984, Sandia National Laboratories, September 2007.

[8] R.A. Bartlett, Teuchos C++ memory management classes, idioms, and related topics: the complete reference (a comprehensive strategy for safe and efficient memory management in C++ for high performance computing), Technical Report SAND2010-2234, Sandia National Laboratories, May 2010, available at: http://www.cs.sandia.gov/ rabartl/TeuchosMemoryManagementSAND.pdf (last accessed 17 April 2012).

[9] V.R. Basili, J.C. Carver, D. Cruzes, L.M. Hochstein, J.K. Hollingsworth, M.V. Zelkowitz and F. Shull, Understanding the high-performance-computing community: a software engineer's perspective, *IEEE Software* **25** (2008), 29–36.

[10] L.S. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley and R.C. Whaley, Practical experience in the dangers of heterogeneous computing, Technical Report UT-CS-96-330, LAPACK Working Note #112, Univ. Tennessee, Knoxville, July 1996,

[11] E. Boman, K. Device, R. Heaphy, B. Hendrickson, W.F. Mitchell, M.S. John and C. Vaughan, *Zoltan: Data-Management Services for Parallel Applications: User's Guide*, available at: http://www.cs.sandia.gov/Zoltan/Zoltan.html (last accessed 17 April 2012), 2004.

[12] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek and S. Tomov, Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy, *ACM Trans. Math. Softw.* **34** (2008), 1–22.

[13] Y. Chen, T.A. Davis, W.W. Hager and S. Rajamanickam, Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate, *ACM Trans. Math. Softw.* **35** (2009), 1–14.

[14] A.T. Chronopoulos and A. Kucherov, Block s-step Krylov iterative methods, *Numer. Linear Algebra Appl.* **17** (2010), 3–15.

[15] J. Daniel, W.B. Gragg, L. Kaufman and G.W. Stewart, Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization, *Math. Comput.* **30** (1976), 772–795.

[16] T.A. Davis, Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method, *ACM Trans. Math. Softw.* **30** (2004), 196–199.

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

[18] T.A. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2006.

[19] T.A. Davis and I.S. Duff, An unsymmetric-pattern multifrontal method for sparse LU factorization, *SIAM J. Matrix Anal. Appl.* **18** (1997), 140–158.

[20] T.A. Davis and I.S. Duff, A combined unifrontal/multifrontal method for unsymmetric sparse matrices, *ACM Trans. Math. Softw.* **25** (1999), 1–19.

[21] T.A. Davis and E. Palamadai Natarajan, Algorithm 907: KLU, a direct sparse solver for circuit simulation problems, *ACM Trans. Math. Softw.* **37** (2010), 36:1–36:17.

[22] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li and J.W.H. Liu, A supernodal approach to sparse partial pivoting, *SIAM J. Matrix Anal. Appl.* **20** (1999), 720–755.

[23] J.W. Demmel, J.R. Gilbert and X.S. Li, An asynchronous parallel supernodal algorithm for sparse Gaussian elimination, *SIAM J. Matrix Anal. Appl.* **20** (1999), 915–952.

[24] J.W. Demmel, L. Grigori, M. Hoemmen and J. Langou, Communication-optimal parallel and sequential QR factorizations: theory and practice, Technical Report UCB/EECS-2008-89, Univ. California Berkeley, 2008, also appears as LAPACK Working Note #204, available at: http://www.netlib.org/lapack/lawns/downloads/.

[25] J. Dongarra, V. Eijkhout and A. Kalhan, Reverse communication interface for linear algebra templates for iterative methods, Technical Report UT-CS-95-291, Univ. Tennessee, Knoxville, May 1995.

[26] R. Falgout, A. Cleary, J. Jones, E. Chow, V. Henson, C. Baldwin, P. Brown, P. Vassilevski and U.M. Yang, Hypre webpage, http://acts.nersc.gov/hypre/ (last accessed 17 April 2012).

[27] B.B. Fraguela, R. Doallo and E.L. Zapata, Memory hierarchy performance prediction for sparse blocked algorithms, *Parallel Process. Lett.* **9** (1999), 347–360.

[28] R.W. Freund, A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems, *SIAM J. Sci. Comput.* **14** (1993), 470–482.

[29] P. Hénon, P. Ramet and J. Roman, PaStiX: a high-performance parallel direct solver for sparse symmetric definite systems, *Parallel Comput.* **28** (2002), 301–321.

[30] M.A. Heroux, AztecOO User Guide, Technical Report SAND2004-3796, Sandia National Laboratories, August 2007.

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

[32] M.R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Natl. Bureau Standards* **49** (1952), 409–436.

[33] Y. Hida, X.S. Li and D.H. Bailey, Algorithms for quad-double precision floating point arithmetic, in: *Proceedings of 15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, Washington, DC, 2001, pp. 155–162.

[34] M. Hoemmen, Communication-avoiding Krylov subspace methods, PhD thesis, EECS Department, Univ. California Berkeley, 2010.

[35] M. Hoemmen, A communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method, in: *Proceedings of 25th IEEE International Parallel and Distributed Processing Symposium*, Anchorage, AK, USA, May 2011.

[36] E.-J. Im, Optimizing the performance of sparse matrix–vector multiplication, PhD thesis, Univ. California, Berkeley, May 2000.

[37] E.-J. Im, K. Yelick and R. Vuduc, Sparsity: optimization framework for sparse matrix kernels, *Int. J. High Perform. Comput. Appl.* **18** (2004), 135–158.

[38] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R.S. Williams and K.A. Yelick, *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*, September 2008. available at: http://users.ece.gatech.edu/~mrichard/ExascaleComputingStudyReports/ECS_reports.htm (last accessed 16 April 2012).

[39] B.C. Lee, R. Vuduc, J.W. Demmel, K.A. Yelick, M. deLorimier and L. Zhong, Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply, Technical Report UCB/CSD-03-1297, Univ. California, Berkeley, November 2003.

[40] X.S. Li and J.W. Demmel, SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems, *ACM Trans. Math. Softw.* **29** (2003), 110–140.

[41] R. Morgan, GMRES with deflated restarting, *SIAM J. Sci. Comput.* **24** (2002), 20–37.

[42] J.J. Navarro, E. García, J.L. Larriba-Pey and T. Juan, Algorithms for sparse matrix computations on high-performance workstations, in: *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PA, USA, May 1996, pp. 301–308.

[43] C.C. Paige and M.A. Saunders, Solution of sparse indefinite systems of linear equations, *SIAM J. Numer. Anal.* **12** (1975), 617–629.

[44] C.C. Paige and M.A. Saunders, LSQR: an algorithm for sparse linear equations and sparse least squares, *ACM Trans. Math. Softw.* **8** (1982), 43–71.

[45] M.L. Parks, E. de Sturler, G. Mackey, D. Johnson and S. Maiti, Recycling Krylov subspaces for sequences of linear systems, *SIAM J. Sci. Comput.* **28** (2006), 1651–1674.

[46] S. Rajamanickam, E.G. Boman and M.A. Heroux, ShyLU: a hybrid–hybrid solver for multicore platforms, in: *Proceedings of 26th International Parallel and Distributed Processing Symposium*, Shanghai, May 2012.

[47] Y. Saad, A flexible inner-outer preconditioned GMRES algorithm, *SIAM J. Sci. Comput.* **14** (1993), 461–469.

[48] Y. Saad and M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.* **7** (1986), 856–869.

[49] M. Sala, K. Stanley and M. Heroux, Amesos: a set of general interfaces to sparse direct solver libraries, in: *Proceedings of PARA'06 Conference*, Umeå, 2006.

[50] M. Sala, K.S. Stanley and M.A. Heroux, On the design of interfaces to sparse direct solvers, *ACM Trans. Math. Softw.* **34** (2008), 1–22.

[51] O. Schenk and K. Gärtner, Solving unsymmetric sparse systems of linear equations with PARDISO, *Fut. Gen. Comput. Syst.* **20** (2005), 475–487.

[52] P. Sonneveld and M.B. van Gijzen, IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations, *SIAM J. Sci. Comput.* **31** (2008), 1035–1062.

[53] K. Soodhalter, Krylov subspace methods with fixed memory requirements: nearly Hermitian linear systems and subspace recycling, PhD thesis, Temple University, 2012.

[54] A. Stathopoulos and K. Wu, A block orthogonalization procedure with constant synchronization requirements, *SIAM J. Sci. Comput.* **23** (2002), 2165–2182.

[55] G.W. Stewart, Block Gram–Schmidt orthogonalization, *SIAM J. Sci. Comput.* **31** (2008), 761–775.

[56] E.D. Sturler, Truncation strategies for optimal Krylov subspace methods, *SIAM J. Numer. Anal.* **36** (1999), 864–889.

[57] S.J. Thomas, A block algorithm for orthogonalization in elliptic norms, in: *Proceedings of the 2nd Joint International Conference on Vector and Parallel Processing*, Lyon, France, September 1–4, Lecture Notes in Computer Science, Vol. 634, Springer, Berlin, 1992, pp. 379–385.

[58] R.S. Tuminaro, M.A. Heroux, S.A. Hutchinson and J.N. Shadid, Official Aztec user's guide, Version 2.1, Technical Report SAND99 8801J, Sandia National Laboratories, November 1999.

[59] H.A. van der Vorst, Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.* **13** (1992), 631–644.

[60] W.A. Wulf and S.A. McKee, Hitting the memory wall: implications of the obvious, *ACM SIGARCH Comput. Architect. News* **23** (1995), 20–24.

[61] F. Xue and H.C. Elman, Fast inexact subspace iteration for generalized eigenvalue problems with spectral transformation, *Linear Algebra Appl.* **435** (2011), 601–622.