Scientific Programming 20 (2012) 257–273
DOI 10.3233/SPR-2012-0347
IOS Press

Playa: High-performance programmable linear algebra

Victoria E. Howle *, Robert C. Kirby, Kevin Long, Brian Brennan and Kimberly Kennedy
Department of Mathematics and Statistics, Texas Tech University, Lubbock, TX, USA
E-mails: {victoria.howle, robert.c.kirby, kevin.long, brian.brennan, kimberly.r.kennedy}@ttu.edu

Abstract. This paper introduces Playa, a high-level user interface layer for composing algorithms for complex multiphysics problems out of objects from other Trilinos packages. Among other features, Playa provides very high-performance overloaded operators implemented through an expression template mechanism. In this paper, we give an overview of the central Playa objects from a user's perspective, show application to a sequence of increasingly complex solver algorithms, provide timing results for Playa's overloaded operators and other functions, and briefly survey some of the implementation issues involved.

Keywords: Numerical linear algebra, high-level software, expression templates

1. Introduction

The Trilinos library provides high-level interfaces to portable, efficient linear and nonlinear algebra. The data structures and algorithms hide the details of distributed memory, and increasingly through Tpetra, shared memory parallelism. Packages such as AztecOO, Belos and Amesos provide robust implementations of iterative and direct linear solver methods, with packages such as Ifpack and ML providing algebraic preconditioners. Building on the successes of these packages for increasingly complex coupled multiphysics problems requires even higher-level abstractions for more involved linear algebraic algorithms. During our work on Sundance [17], a high-level Trilinos package for automating the construction of finite element operators, we have found need of many extended capabilities built on top of these packages. These capabilities have been put together in a new Trilinos package *Playa*.

Playa lakes are novel geographical features of the South Plains of eastern New Mexico and West Texas. They are shallow, often seasonal, lakes supporting a diverse ecosystem that includes native and migratory species. In the same way, we hope to provide a thin layer of support on top of existing Trilinos packages to support new algorithmic research on coupled problems, optimization, and other challenging topics.

*Corresponding author: Victoria E. Howle, Department of Mathematics and Statistics, Texas Tech University, Lubbock, TX 79409, USA. E-mail: victoria.howle@ttu.edu.

1.1. Design example

We begin with a short code example. Many of the Playa features evident in this example will be described later in the paper. The purpose here is simply to illustrate some of the important features of Playa for potential end users. In this example, we show a simple implementation of the Conjugate Gradients algorithm in Playa. For simplicity and brevity, preconditioning and error checking are omitted, and parameters such as the stopping tolerance and maximum number of iterations are hard coded. The human readability of Playa code can be seen by comparing this code to a standard pseudo-code statement of the Conjugate Gradients algorithm such as that in [20], Algorithm 38.1.

Upon exiting the CG loop, the solution is in the x vector. An industrial-strength CG solver would certainly need preconditioning and error checking, and could be packaged up as a `LinearSolver` object. But this example illustrates the clean Playa interface.

1.2. Design goals

Frequently, even successful HPC codes start from a bottom-up approach that emphasizes performance without significant consideration for higher-level programmability. This complicates research and development of algorithms that do not map into simple combinations of library calls. In Playa, we have started with a top-down design that aims to combine programmabil-

1058-9244/12/$27.50 © 2012 – IOS Press and the authors. All rights reserved

Code Example 1. Simple implementation of unpreconditioned CG in Playa

```
Vector<double> x = b.copy(); // deep copy
Vector<double> r = b − A∗x;
Vector<double> p = r.copy();
Vector<double> Ap = A∗p;

double tol = 1.0e−12;
int maxIter = 100;

for (int i=0; i<maxIter; i++)
{
  double rSqOld = r∗r; // dot product
  double pAp = p∗Ap;
  double alpha = rSqOld/pAp;

  x += alpha∗p; // x = x + alpha∗p
  r −= alpha∗Ap; // r = r − alpha∗Ap

  double rSq = r∗r; // dot product
  double rNorm = sqrt(rSq);
  Out::root() << "iter=" << setw(6) << i
  << setw(20) << rNorm << endl;

  if (rNorm < tol) break;

  double beta = rSq/rSqOld;
  p = r + beta∗p;
  Ap = A∗p; // save this; we'll use it twice
}
```

ity and flexibility on top of existing high-performance libraries.

We stress interoperability of vectors, matrices, and solver libraries via a `VectorType` class. Also, on top of this interoperability framework lies a handle layer that manages memory without requiring the user to explicitly manipulate pointers or some kind of smart pointer. The high-level syntax afforded by this handle layer allows us to present an efficient expression template engine [13,21] that not only performs standard matrix and vector operations, but also supports deferred evaluation of transposition and inversion (through an existing linear solver).

This interoperability is also featured in some existing Trilinos packages. While the Thyra library [4] provides a relatively low-level interoperability suite between different code types, Playa targets the user-level experience. Also aiming for generality, Belos provides Krylov methods for very general kinds of operators via traits. We aim at a similar level of flexibility, although are oriented towards general programmability rather than optimally architecting Krylov methods. We provide a compatibility layer so that very complex, implicitly-defined Playa operators may be efficiently used within Belos solvers.

Another goal is the support of coupled systems that arise in multiphysics applications. Such problems frequently give rise to linear systems that have a block structure, in which operators are themselves arrays of operators. In some cases, this blocking can be recursive (multiply-layered). Playa provides extensive support for this case, and we will present examples of forming (implicit) Schur complements and block structured preconditioners for some model problems in fluid mechanics.

Besides these features, we provide a mechanism by which various Trilinos solvers are united in a common framework and specified at run-time through a `ParameterList`. This allows solver decisions to be deferred to run-time, and users have access to a wide range of Trilinos packages via XML files without having to explicitly remember the calling conventions within C++. Currently solvers from Amesos, Belos and AztecOO are supported, as are preconditioners from Ifpack and ML. We have found this ability to switch between solvers without penalty or need to write or compile new C++ quite useful.

1.3. Related software

Playa shares similarities with several existing packages. Within Trilinos, Thyra [4] also provides interoperability between various Trilinos packages, blocked operators, and various utilities. Relative to Thyra, Playa's class hierarchy is sparser. Also, the handle layer with expression templates on top is distinctive, and our interoperability mechanisms, such as abstract interfaces for loading vectors and matrices, should allow Playa to be more easily extended to support additional linear algebra packages in the future. Also within Trilinos, Teko [19] contains similar features for blocking and some advanced utility features for constructing block preconditioners, although their focus is more limited to block-structured solver research. High-level PDE packages such as DOLFIN [15] and Deal.II [2] also provide some kind of interoperability, although these packages essentially wrap PETSc and Trilinos, along with other libraries, behind a common abstract API. Playa provides a high-level architecture for linear algebra, while aiming at reuse of underlying libraries.

1.4. Organization of this paper

In Section 2, we first give a user-level introduction to some of the key features of Playa, including handles and memory management through reference-counted pointers and an overview of some of the principal user-level classes. In Section 3, we walk through a few relatively simple examples in Playa with a view to giving the users a more concrete sense of the advantages of Playa. These advantages may not be so appealing if they came at the price of efficiency. In Section 4, we provide some timing results comparing Playa and Epetra on a range of different platforms to demonstrate that our high-level interface does not degrade the performance of basic linear algebra operations. Finally, in Section 5, we provide more detailed implementation information. These details are critical to the efficiency and the ease of use of Playa, and some users will want to know these details and understand how Playa achieves efficiency. However, understanding these implementation issues for the typical user to effectively use Playa.

2. User-level view of Playa

In this section we survey the core Playa object suite from a user's point of view, deferring most implementation details until Section 5.

Code fragments and references to class names will be written in `typewriter` font. For brevity and clarity, class names used in English sentences rather than code will often have template arguments suppressed where convenient, for example, although Playa vector objects are templated on scalar type we will often speak of a `Vector` instead of a `Vector<Scalar>`.

2.1. Memory management: Reference-counted pointers and handles

One implementation issue that should be discussed early is memory management. Playa makes extensive use of the templated reference-counted pointer (RCP) tools available through Teuchos [3]. Consequently, in any standard use case, the user will have transparent, safe, and robust memory management ensured when using Playa objects.

Playa further hides memory management issues from the user by wrapping RCPs in handle classes, so that typical Playa objects will have value syntax rather than pointer syntax. We'll refer to RCP-based handles as reference-counted handles, or RCH. Beyond the user-level convenience of value syntax, RCH also provide a common point of entry for certain maintenance tasks that should be done neither by the user nor by concrete instances of derived types. These issues will be discussed in more detail below, after the relevant classes have been introduced. One consequence of the use of reference-counting is that by default, copies of Playa objects are shallow. That means that an assignment such as

```
Vector<double> y = x;
```

does not create a new copy of the vector x, complete with new data. Rather, it creates a new RCH to the same data. One advantage of this is obvious: vectors can be large, so we want to avoid making unnecessary copies. But note that any modification to y will also trigger the same modification to x, because x and y are referring to *exactly the same data in memory.* The potential for confusion and unintended side effects is obvious. Less obvious is that in certain important circumstances, such side effects are exactly what is needed for a clean user interface to efficient low-level code. Deep copies require use of the `copy()` member function.

2.1.1. Issues with handles and polymorphism

An issue with handles to polymorphic hierarchies is the decision of which member functions of possible subtypes are to be propagated to the handle-level interface. For example, as will be discussed in more detail below, vectors can have block structure.

One resolution to this issue is to force explicit user-level dynamic casts to specialized subtypes. Another is to propagate specialized methods up to the handle layer. In the design of Playa, we try to avoid user-level dynamic casts whenever possible. Our rough guideline for deciding which methods appear in the common user interface is that implementation-dependent methods (such as a method specific to a particular vector representation such as Epetra) should not appear in the common interface, however, methods specific to a mathematically-distinct type of object (such as a block vector) can. Dynamic casts must still be done, of course, but are done by the handle layer itself; this both hides such issues from the user and provides a common point of entry for error detection and handling.

One additional issue arises with error handling and dynamic typing. The dynamic polymorphism used in Playa requires run-time error handling. While many other portions of Trilinos rely on compile-time error checking, this comes at the expense of a much larger type system that some users find difficult to navigate.

2.2. Key classes

The principal user-level classes in Playa are listed here:

- `Vector` objects represent mathematical vectors. In addition to the fundamental operations of vector addition and scalar–vector multiplication, arbitrary user-defined transformation and reduction operations can be implemented through application of templated functors.
- `LinearOperator` objects represent linear functions that map vector inputs to vector outputs.
- `LinearSolver` objects represent algorithms for solving linear systems.
- `VectorSpace` is an abstract factory that produces Vector objects. This provides a consistent user interface for creating vectors without the client code needing to know anything about what sort of vector object is being created.
- `VectorType` is an abstract factory that produces VectorSpace objects of a user-specified type, dimension, and internal data layout. A client such as a finite element code uses a VectorType to create a VectorSpace once the dimension and distribution of its discrete space has been determined.

Each of these five classes is a RCH to an underlying base class of the appropriate type: Vector is a RCH to a VectorBase and so on.

2.3. Vector spaces

A user will rarely construct a vector directly; the reason for this is that different vector implementations have different data requirements making it difficult to provide a uniform vector constructor; furthermore, vectors are often created inside solver algorithms hidden from the solver rather than in user-space code such as `main()`. Polymorphic object creation is a common design problem with a common solution: the factory class [9]. The realization that a *vector space object* could function as a factory class for vectors was due to Gockenbach and Symes [10] who incorporated it in their influential Hilbert Class Library (HCL). Descendents of the HCL including TSF [16], TSFCore [5], Thyra [6] and now Playa, have continued the use of this pattern.

With this design pattern, vectors are built indirectly by calling the `createMember()` member function of `VectorSpace`. Each `VectorSpace` object contains the data needed to build vector objects, and the implementation of the `createMember()` function will use that data to invoke a constructor call.

Every `Vector` retains a RCH to the space that created it. Among other things, this can be used for checking compatibility in vector–vector and operator–vector operations. To support such compatibility checking, the `VectorSpace` class provides overloaded `==` and `!=` operators as member functions.

The `VectorSpace` class also provides member functions that describe block structure, element indexing and data layout.

2.3.1. Block structure

In optimization and physics-based preconditioning, it is common to segregate variables into blocks. Mathematically, this is a factoring of a problem's vector space setting into a Cartesian product of smaller spaces. Segregation into blocks can be done recursively; i.e., each block of a block space could itself be a block space. Iteration over blocks requires a means of indexing into a space's block structure. This can be done in several ways. The simplest is to identify a block by its index into the current level, that is, without recursion into sub-block structure. To identify a block at an arbitrary depth, a double-ended queue (deque) of indices can be used. Finally, the index deque can be encapsulated in a `BlockIterator` object, allowing in-order traversal of any arbitrary block structure in a single loop.

The user interface of the `VectorSpace` object provides member functions that describe that space's block structure, and also a member function that creates a `BlockIterator` appropriate to that space. A `BlockIterator` can then be used to identify blocks in any `Vector` created by the space in question, or block rows (or columns) of any `LinearOperator` having the space in question as its range (or domain).

2.4. Vectors

`Vector` is the user-level vector object. As such, it supports through member functions many commonly-used mathematical operations such as linear combinations, dot products, and norms. Adding new vector operations in a scalable way is discussed in Section 5.

2.4.1. Overloaded operations

Whenever possible, Playa uses overloaded operators to represent vector operations; for example, the opera-

tion

$$x = x + \beta y + \gamma z,$$

where x, y, z are vectors and β, γ scalars would be written

```
x += beta*y + gamma*z;
```

using overloaded scalar–vector multiplication, vector addition and add-into operators. It is well known that naive implementations of overloaded vector operations are unacceptably slow due to creation of temporary objects (see, e.g., [18]); nonetheless, efficient operator overloading implementations have been devised [13, 22]. Our fast implementation of overloaded operations on polymorphic vectors will be described in Section 5, and timings demonstrating negligible performance penalty will be shown in Section 4.

2.4.2. Access to vector data

Access to vector data is desired in several contexts. First and most importantly, during operations such as the fill step in a finite element code, a simulator will write data into specified elements in a vector, and in coefficient evaluation the same simulator will read field data stored in a vector. A complicating factor in the read use case is the need to access nonlocal "ghost" elements. In Playa, these operations are not part of the `Vector` interface; rather, specialized `LoadableVector` and `GhostView` interfaces are used. The interface between vector and simulator will not be discussed further in this paper.

A second use case for data access is the occasional desire for quick and simple access to a vector element through an operation such as

```
cout << ``x[3] = `` << x[3] << endl;
```

Universally efficient implementation of such "bracket" operators is not possible with polymorphic vector types, so it is rarely a good idea to program mathematical operations through user-level element access. However, for debugging and prototyping, it is often most efficient *in terms of programmer time* to have such operations available; therefore, Playa supports element read and write through bracket operators. The indices used to specify elements here are always local to a given processor. In the case of block vector spaces, a natural local indexing for the entire block vector is defined recursively using the block dimensions and the local indexing in the blocks.

2.5. Operators

The `LinearOperator` RCH is the user-level class representation of linear operators, including not only various matrix representations (including block matrices) but also implicit operators whose action on a vector is to be computed without explicit construction of a matrix representation.

The overloaded $+$, $-$, $*$ operators have their conventional meaning of addition, subtraction, and composition of operators, respectively, but those operations are done in an implicit sense. The action of a composed operator ABx is computed implicitly by first computing $y = Bx$, then computing Ay. There is no need to form the matrix AB. Similarly, $(A \pm B)x$ can be evaluated implicitly by computing $y = Ax$, $z = Bx$, then doing $y \pm z$. Action of a scaled operator αAx is done implicitly as $\alpha(Ax)$. Any combination of these can be specified using overloaded operators, for example:

```
LinearOperator<double> C = A + B;
LinearOperator<double> D = 2.0*A - 0.5*B + 1.2*C;
LinearOperator<double> E = A*B;
```

Certain simple operators have implicit representation. A diagonal operator can be represented with nothing but a vector of diagonal elements. Application of the zero operator returns the zero vector of the range space of the operator. The identity operator simply returns a copy of the operand. Playa provides functions to construct all of these simple operators.

Finally, it is often useful to have implicit object representations of transposes and inverses. Most good sparse matrix packages have the ability to compute $A^{\mathrm{T}}x$ without explicitly forming A^{T}. Given that, together with implicit composition, we can do $(AB)^{\mathrm{T}}x = B^{\mathrm{T}}A^{\mathrm{T}}x$ implicitly as well, and with implicit addition we can do $(A \pm B)^{\mathrm{T}}x = A^{\mathrm{T}}x \pm B^{\mathrm{T}}x$. The `transpose()` member function creates an operator object that knows to apply these rules. Note that the `transpose()` function is a member function of the `LinearOperator` class and not of the `Vector` class. Playa makes no distinction between row and column vectors, so `transpose()` does not make any sense in the context of `Vector` objects. An inner product of two vectors `u` and `v`, for example, would be written as `u*v`.

The operation $y = A^{-1}x$ is computed implicit by solving the system $Ay = x$. It is necessary to specify the solver algorithm that will be used to solve the system, by means of providing a `LinearSolver` object

as an argument to the inverse function.

```
LinearOperator<double> AInv = inverse(A, solver);
```

The operation of solving $Ax = b$ for x can then be encapsulated nominally as multiplication $x = A^{-}1b$. This notational simplification is important in composing algorithms such as block preconditioners, where the application of a preconditioner might involve several solves on subsystems.

Each `LinearOperator` has `VectorSpace` members indicating the domain and range of the operator. Block structure in the domain and/or range is naturally propagated to the operators. These blocks may be manipulated with basic `getBlock()` and `setBlock()` operations, and a nonmember `makeBlockOperator()` methods produces empty block operators for user-defined construction.

2.6. Linear solvers

Linear solvers are represented by the `LinearSolver` RCH. The most often used member function of `LinearSolver` is `solve()`. Examples of solver subtypes include wrappers for Amesos, Aztec and Belos solvers. It is also easy to write new solvers using Playa objects; several examples of this will be shown in Section 3.

Not every solver algorithm will be compatible with every operator instance; for example, an implicit inverse operator will not normally be able provide the elementwise information needed for use in a direct solver. Such incompatibilities are checked for at runtime, and if encountered, an exception is thrown.

2.6.1. Preconditioners and preconditioner factories

Playa's representation of a preconditioner is an object that contains two operators that can be applied as left and right preconditioners. Also necessary are introspection functions that indicate whether a given preconditioner has left and right operators.

Creation of actual preconditioner objects is usually not done directly by the user, but indirectly by a solver: a natural use case for the factory pattern. Consequently, the user will normally specify a `PreconditionerFactory` object which when given a `LinearOperator` instance, can build a `Preconditioner` object of the appropriate type.

Examples of simple Playa preconditioner factories include factories to construct incomplete factorization preconditioners through Ifpack and algebraic multigrid preconditioners through ML. Several examples of building physics-based block preconditioners are shown in the examples in Section 3.

2.7. Key concrete adapters

To use the high-level capabilities of Playa to drive calculations with a specified low-level data representation (e.g., Epetra), one must write a small set of adapter classes that implement the Playa `VectorTypeBase`, `VectorSpaceBase` and `VectorBase` interfaces.

At present the workhorse of Trilinos-based simulation is still Epetra, and so the most widely-used Playa concrete adapters are for Epetra objects. We also provide adapters for serial dense matrices and vectors. The abstraction is general enough to incorporate Tpetra operators and vectors as well as other libraries such as PETSc.

3. Playa examples

In this section, we highlight some of the important features of Playa through a few more examples starting with the power method and inverse power method, then progressing to the solution and preconditioning of the linear systems arising in the incompressible Navier–Stokes equations and preconditioners for this problem, and finally to a more complicated block preconditioner for a coupled fluid-thermal problem.

3.1. Power iteration and inverse power iteration

We define a simple function implementing the power method for calculating the largest eigenvalue and corresponding eigenvector of a matrix A. With polymorphic operator objects, a power method code can be converted to an inverse power method code simply by calling the method with A^{-1} rather than A. In Code Example 2, we call the power method with a linear operator A to calculate its largest eigenvalue and eigenvector and with an implicit A^{-1}, resulting in an application of the inverse power method and returning the reciprocal of the lowest eigenvalue and corresponding eigenvector of A. The implicit inverse operator evaluates $A^{-1}y$ using a `LinearSolver` to solve the system $Ax = y$. The implementation of the power iteration method is shown in Code Example 3.

This example, like the Conjugate Gradients example in the Introduction, has concentrated on the *use* of matrix and vector objects rather than the *creation* of these objects, since they will be built by an application code. However, in writing advanced preconditioning, optimization, and solver algorithms, composing implicit operations will be critical.

Code Example 2. Power method and inverse power method in Playa

```
// A is a LinearOperator previously defined.
// x is a Vector previously defined.

// Power method with A
double lambda_largest = powerMethod(A,x);

// Inverse power method using implicit A inverse
LinearSolver<double> solver =
  LinearSolverBuilder::createSolver("amesos.xml");
LinearOperator<double> Ainv = inverse(A,solver);
double lambda_smallest =
  (1.0 / powerMethod(Ainv,x));
```

Code Example 3. Power method in Playa

```
double powerMethod(const LinearOperator<double>&
   A, Vector<double>& x)
{
  // Normalize initial ev guess x.
  x = 1.0 / x.norm2()*x;

  // Set parameters for power method.
  int maxIters = 500;
  double tol = 1.0e-10;
  double mu;
  double muPrev;

  for (int i=0; i<maxIters; i++)
  {
    Vector<double> Ax = A*x;
    mu = (x*Ax) / (x*x);
    cout << "Iteration " << i << " mu = "
         << setprecision(10) << mu << endl;
    if (fabs(mu - muPrev) < tol) break;
    muPrev = mu;
    double AxNorm = Ax.norm2();
    x = 1.0 / AxNorm*Ax;
  }
  return mu;
}
```

3.2. Incompressible Navier–Stokes

The previous sections illustrated some simple applications of Playa and its potential for ease of programmability. Now we combine these tools with block-structured operations to develop complex preconditioners using incompressible Navier–Stokes as an example.

3.2.1. Schur complement

Using Playa's blocking and expression template capabilities, it is straightforward to form Schur complements of matrices. In this example, we solve the incompressible Navier–Stokes equations via the Schur complement. Consider the incompressible Navier–Stokes equations,

$$-\nu\Delta u + u\cdot\nabla u + \nabla p = f, \qquad \nabla\cdot u = 0, \quad (1)$$

posed on some domain $\Omega \subset \mathbb{R}^d$ for $d = 2, 3$ and equipped with appropriate boundary conditions. Here, u is the d-dimensional velocity field, p is the pressure, and ν is the kinematic viscosity, which is inversely proportional to the Reynolds number.

Linearization and (*Div-stable*) discretization of (1) leads to a linear system of the form

$$\begin{pmatrix} F & B^{\mathrm{T}} \\ -B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, \quad (2)$$

where B and B^{T} are matrices corresponding to discrete divergence and gradient operators, and F operates on the discrete velocity space.

Key in a Schur complement solve is the use of an implicit inverse for F^{-1} and deferred evaluation of the product $BF^{-1}B^{\mathrm{T}}$. These features are illustrated in Code Example 4. In this example, we assume that the `LinearProblem prob` has been previously defined with Sundance, the operator K is a block 2×2 linear operator as in (2), and the solver for the F matrix has been defined elsewhere in Sundance. The linear systems can be written as the following Schur complement system

$$(BF^{-1}B^{\mathrm{T}})p = BF^{-1}f_1 + f_2, \quad (3)$$

which we solve for p. The current velocity u is then formed from the system

$$Fu = f_1 - B^{\mathrm{T}}p. \quad (4)$$

In Code Example 4, we build the Schur complement operator making use of deferred evaluation and implicit inverses. This code fragment would be part of a nonlinear solver loop such as a Newton or a fixed point iteration.

3.2.2. PCD block preconditioner

We can also form complicated block preconditioners such as the pressure convection–diffusion (PCD) pre-

Code Example 4. Schur complement solve in Playa

```
// The LinearProblem prob has already been defined
// in Sundance.
// K is the block 2x2 operator and rhs is the
// blocked right-hand side from equation (2).
LinearOperator<double> K = prob.getOperator();
Vector<double> rhs = prob.getSingleRHS();

//Extract the subblocks from K to use in forming S
LinearOperator<double> F = K.getBlock(0,0);
LinearOperator<double> Bt = K.getBlock(0,1);
LinearOperator<double> B = K.getBlock(1,0);

// Define an implicit inverse on F. The
// LinearSolver FSolver
// has been defined outside the nonlinear loop.
LinearOperator<double> FInv = inverse(F, FSolver);

// Build the Schur complement (deferred
// evaluation).
LinearOperator<double> S = B*FInv*Bt;

// Get the RHS subblocks for the Schur solve
Vector<double> urhs = rhs.getBlock(0);
Vector<double> prhs = rhs.getBlock(1);

// Build the RHS for the Schur complement system.
Vector<double> yp = B.range().createMember();
yp = prhs + B*FInv*urhs;

// Solve the Schur complement system for pressure.
// The Schur complement LinearSolver has been
// defined elsewhere.
Vector<double> pnew = B.range().createMember();
SolverState<double> SState =
   SSolver.solve(S, yp, pnew);

// Calculate the new velocity as in equation (4).
Vector<double> uNew = B.domain().createMember();
uNew = FInv * (urhs - Bt * pnew);
```

conditioner [8,12] for solution of the incompressible Navier–Stokes equations. This preconditioner has the block form

$$P_N^{-1} = \begin{pmatrix} F^{-1} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & -B^{\mathrm{T}} \\ 0 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \tilde{S}^{-1} \end{pmatrix} \tag{5}$$

which results from a block LU factorization of the matrix in (2). Here $\tilde{S}$ is an approximation to the Schur complement S. In PCD, we have $\tilde{S}^{-1} = M_{\mathrm{p}}^{-1} F_{\mathrm{p}} A_{\mathrm{p}}^{-1}$, where A_{p} is a discrete Laplacian operator on the pressure space, F_{p} is a discrete convection–diffusion operator on the pressure space, and M_{p} is a pressure mass matrix. The operators A_{p}, F_{p} and M_{p} are needed by the PCD preconditioner from the user; we can easily generate these operators with Sundance. An application of the preconditioner thus requires linear solves with the operators A_{p}, M_{p} and F.

Playa's support of block structure, deferred evaluation, and implicit inverses lead to relatively simple code for this complicated block preconditioner. The resulting Playa code looks very much like "hand-written" equations. The PCD preconditioner has been implemented in Playa and derives from the `PreconditionerFactory` class. Code Example 5 illustrate these features.

Since the Navier–Stokes equations are nonlinear, we need a nonlinear solver in this PCD example and in the previous Schur complement example. We used Newton's method in our implementations. One can code the Newton system directly or use the nonlinear operator class `NLOp` in Sundance. The `NLOp` class encapsulates a discrete nonlinear problem and can be passed to a nonlinear solver such as NOX [14].

3.3. Bénard convection

We conclude this section with an even more complicated block preconditioner developed for a coupled fluid-thermal problem. Consider the Bénard convection problem

$$\begin{aligned} -\Delta u + u \cdot \nabla u + \nabla p &= -\frac{Ra}{Pr} \hat{g} T, \\ \nabla \cdot u &= 0, \\ -\frac{1}{Pr} \nabla T + u \cdot \nabla T &= 0, \end{aligned} \tag{6}$$

posed on some domain Ω along with boundary conditions, where the Rayleigh number *Ra* measures the ratio of energy from buoyant forces to viscous dissipation and heat conduction, the Prandtl number *Pr* measures the ratio of viscosity to heat conduction, and $\hat{g}$ denotes a unit vector along the axis in which gravity acts.

Linearizing at the level of the weak form, gives rise to a linear variational problem for the Newton step for Navier–Stokes. The linear system for a Newton step for the convection problem takes the form

$$\begin{pmatrix} F & B^{\mathrm{T}} & M_1 \\ -B & 0 & 0 \\ M_2 & 0 & K \end{pmatrix} \begin{pmatrix} u \\ p \\ T \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}, \tag{7}$$

Code Example 5. PCD preconditioner in Playa

```
// LinearSolvers and the LinearProblems to form
// Fp, Ap and Mp
// were defined when the PCDPreconditionerFactory
// was constructed.
// This code takes the 2x2 block operator K of
// equation (2)
// and returns a PCD right preconditioner.
Preconditioner<double>
PCDPreconditionerFactory::
createPreconditioner(const LinearOperator<double>&
    K) const
{
  // Get subblocks from K and Fp, Mp and Ap linear
  // operators
  // for PCD. Set up implicit inverses on F, Mp
  // and Ap.
  LinearOperator<double> F = K.getBlock(0,0);
  LinearOperator<double> FInv =
      inverse(F,FSolver_);
  LinearOperator<double> Bt = K.getBlock(0,1);
  LinearOperator<double> Fp =
      FpProb_.getOperator();
  LinearOperator<double> Mp =
      MpProb_.getOperator();
  LinearOperator<double> MpInv =
      inverse(Mp, MpSolver_);
  LinearOperator<double> Ap =
      ApProb_.getOperator();
  LinearOperator<double> ApInv =
      inverse(Ap, ApSolver_);

  // Build identity operators.
  LinearOperator<double> Iu =
      identityOperator(F.domain());
  LinearOperator<double> Ip =
      identityOperator(Bt.domain());

  // Build PCD approximation to inverse Schur
  // complement.
  // Using deferred evaluation and implicit
  // inverses.
  LinearOperator<double> XInv = MpInv*Fp*ApInv;

  // Make three 2x2 block operators for the PCD
  // preconditioner
  VectorSpace<double> rowSpace = K.range();
  VectorSpace<double> colSpace = K.domain();
  LinearOperator<double> Q1 = makeBlockOperator(
      colSpace, rowSpace);
  LinearOperator<double> Q2 = makeBlockOperator(
      colSpace, rowSpace);
  LinearOperator<double> Q3 = makeBlockOperator(
      colSpace, rowSpace);

  // PCD continued:

  // Populate the three block operators that make
  // up the PCD preconditioner.
  Q1.setBlock(0, 0, FInv);
  Q1.setBlock(1, 1, Ip);
  Q1.endBlockFill();
  Q2.setBlock(0, 0, Iu);
  Q2.setBlock(0, 1, -1.0*Bt);
  Q2.setBlock(1, 1, Ip);
  Q2.endBlockFill();
  Q3.setBlock(0, 0, Iu);
  Q3.setBlock(1, 1, -1.0*XInv);
  Q3.endBlockFill();

  // The PCD preconditioner is the product of
  // these three 2x2 systems.
  LinearOperator<double> PInv = Q1 * Q2 * Q3;

  // Return a PCD right preconditioner.
  return new GenericRightPreconditioner<double>
      (PInv);
}
```

where the matrices F and B are the same as in linearized Navier–Stokes (2). The matrix M_1 arises from the term $(\frac{Ra}{Pr}T, v_g)$, where g is the Cartesian direction in which gravity acts (y in 2D, z in 3D). In 2D, if the velocity variables in each direction are segregated, this has the form of $M_1^{\mathrm{T}} = (0, M^{\mathrm{T}})$, where M is a rectangular mass matrix. The matrix M_2 arises from the Jacobian term $(u \cdot \nabla T_0, r)$, where T_0 is the temperature in the current Newton iterate. The matrix K comes from $\frac{1}{Pr}(\nabla T, \nabla r) + (u_0 \cdot \nabla T, r)$, so it is a standard linear convection–diffusion operator.

A recently developed block preconditioner for this system [11] is given by

$$P^{-1} = \begin{pmatrix} N^{-1} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} I & -\tilde{M}_1 \\ 0 & I \end{pmatrix} \times \begin{pmatrix} I & 0 \\ 0 & K^{-1} \end{pmatrix}, \tag{8}$$

where N is the 2×2 block system from the Navier–Stokes equations (2). Applying the preconditioner requires the (preconditioned) solution of the linearized Navier–Stokes system N and of the scalar convection–diffusion system K.

Implementation of this preconditioner in Playa uses many of the same techniques as described in the pre-

vious examples. In particular, we make use of deferred evaluation, implicit inverses, and block operators. For this preconditioner, there is the added complication of a nested block structure. The $(0,0)$ block of the preconditioner (8) is itself a block operator. Playa supports nested block structure, so implementation of this preconditioner was straightforward. In Code Example 6, we show a fragment of code for building the nested 2×2 operators for the preconditioner.

Code Example 6. Nested blocks in Playa

```
// Set up nested block operator given subblock
// components (already built).

// Make spaces for inner 2x2 block system.
int nBlocks = 2;
Array<VectorSpace<double>> space2x2inner(nBlocks);
space2x2inner[0] = F.domain();
space2x2inner[1] = Bt.domain();
VectorSpace<double> blkSp2x2inner =
   blockSpace(space2x2inner);
LinearOperator<double> N = makeBlockOperator(
     blkSp2x2inner, blkSp2x2inner);
N.setBlock(0,0,F);
N.setBlock(0,1,Bt);
N.setBlock(1,0,B);
N.setBlock(1,1,C);
N.endBlockFill();

// Make 2x2 Jacobian matrix [N MM1; MM2 K]
Array<VectorSpace<double>> space2x2outer(nBlocks);
space2x2outer[0] = N.domain();
space2x2outer[1] = M1.domain();
VectorSpace<double> blkSp2x2outer =
   blockSpace(space2x2outer);
LinearOperator<double> blockJ = makeBlockOperator(
     blkSp2x2outer,blkSp2x2outer);
LinearOperator<double> MM1 = makeBlockOperator(M1.
     domain(),blkSp2x2inner);
LinearOperator<double> MM2 = makeBlockOperator(
     blkSp2x2inner,M1.domain());
MM1.setBlock(0,0,M1);
MM1.endBlockFill();
MM2.setBlock(0,0,M2);
MM2.endBlockFill();
blockJ.setBlock(0,0,N);
blockJ.setBlock(0,1,MM1);
blockJ.setBlock(1,0,MM2);
blockJ.setBlock(1,1,K);
blockJ.endBlockFill();
// Nested Blocks in Playa (cont.)

// Make nested 2x2 preconditioner operators

// P^{-1} = [Ninv 0; 0 I] [I -MM1; 0 I] [I 0; 0
// Kinv]
LinearOperator<double> Iu =
   identityOperator(F.domain());
LinearOperator<double> Ip =
   identityOperator(Bt.domain());
LinearOperator<double> It =
   identityOperator(M1.domain());
LinearOperator<double> Iup =
   identityOperator(N.domain());

LinearOperator<double> Ninv =
   inverse(N, NSSolver);
LinearOperator<double> P1 = makeBlockOperator(
     blkSp2x2outer,blkSp2x2outer);
LinearOperator<double> P2 = makeBlockOperator(
     blkSp2x2outer,blkSp2x2outer);
LinearOperator<double> P3 = makeBlockOperator(
     blkSp2x2outer,blkSp2x2outer);

P1.setBlock(0,0,Ninv);
P1.setBlock(1,1,It);
P1.endBlockFill();

P2.setBlock(0, 0, Iup);
P2.setBlock(0, 1, -1.0*MM1);
P2.setBlock(1, 1, It);
P2.endBlockFill();

P3.setBlock(0, 0, Iup);
P3.setBlock(1, 1, Kinv);
P3.endBlockFill();

LinearOperator<double> Pinv = P1*P2*P3;
```

4. Some basic timing results

In order to demonstrate that our high-level interface does not degrade the performance of basic linear algebra operations, we have tested some operations common to Playa and Epetra on a range of different platforms. Playa relies on Epetra (and, in the future, potentially other vector types such as Tpetra or PETSc) to provide data containers, and we can either forward the evaluation of our expression templates to the underlying Epetra function or else stream functors over the data provided by Epetra. In either case, our baseline to compare against is Epetra – apart from O(1) overhead, Playa performance should match that of the underlying vector library.

We consider three operations: the dot product of two vectors, computation of the vector 1-norm, and evaluation of a linear combination of two vectors. The first

two of these are reduction operations, mapping the input vector(s) to a single number. The third example maps vectors to vectors. In our timing results, the Playa implementation of the dot product simply forwards the function call to Epetra and reports the result. On the other hand, the vector 1-norm is evaluated by means of streaming functors over data chunks. In other words, Epetra provides a data container but no arithmetic. Finally, the linear combination is evaluated by means of the underlying Epetra operation. In each case, we repeated the calculation 100 times and report the average time.

We also tested our operations on several UNIX-like architectures. We ran tests in serial on the following platforms:

(a) Mac Pro desktop (dual quad-core 2.8 GHz Xeon processors with 32 GB of RAM) running OSX version 10.5, with the code compiled using gcc version 4.4.4 installed from the MacPorts system.
(b) Dell Inspiron 1120 laptop with two 1.3 GHz AMD Athlon Neo K325 dual-core processors and 4GB of RAM running Ubuntu Linux and gcc 4.5.2.
(c) a Dell Precision M6500 laptop with dual 2 GHz quad-core i7 processors and 16 GB of RAM running gcc version 4.4.3 on Ubuntu 10.04.
(d) a HP dual quad-core i7 machine with 16 GB of RAM, compiled under gcc 4.5.2 on Ubuntu Linux.

Figures 1–4 show mostly what we expect. Although the basic FLOP rates vary from machine to machine, we notice that Playa has an additional cost relative to Epetra for small vectors, but that for moderate-sized vectors the performance is identical. On the other hand, the performance of the vector 1-norm, computed with

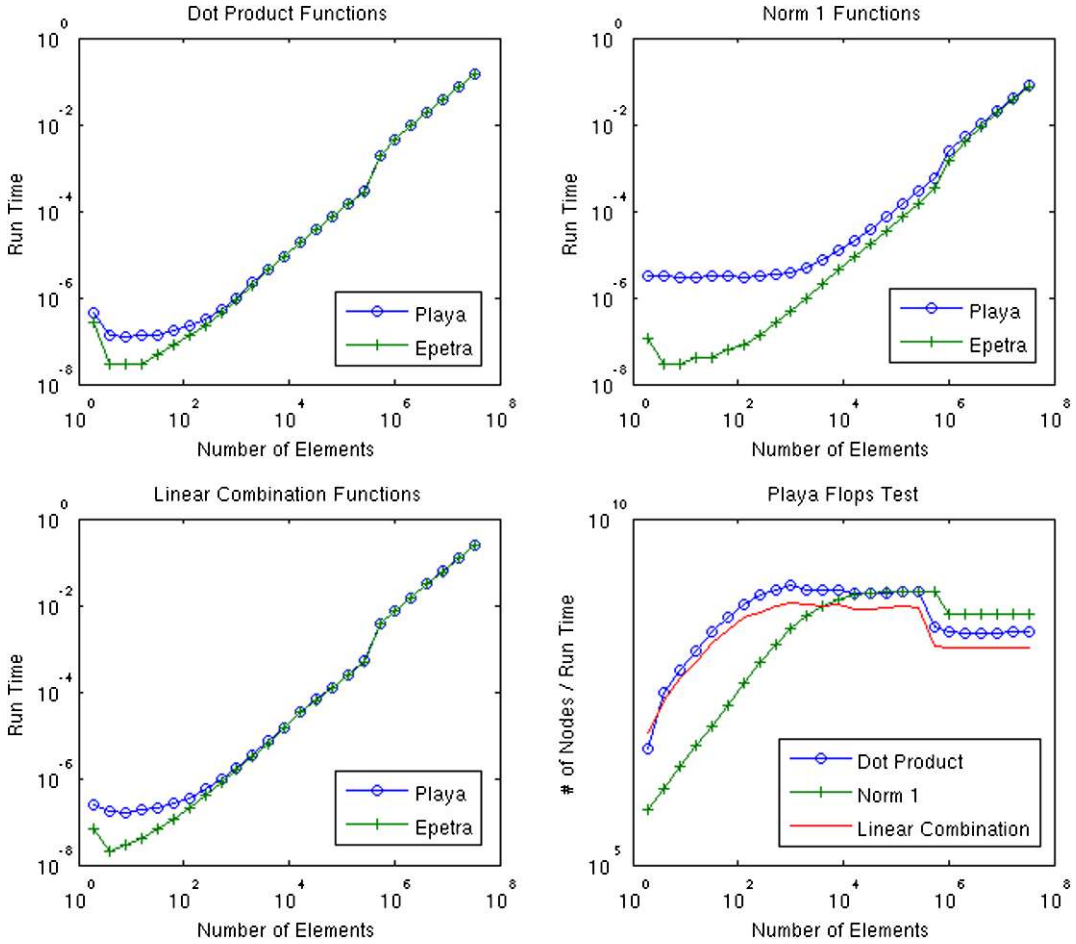


Fig. 1. Comparison of Playa and Epetra performance on platform (a). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0347.)

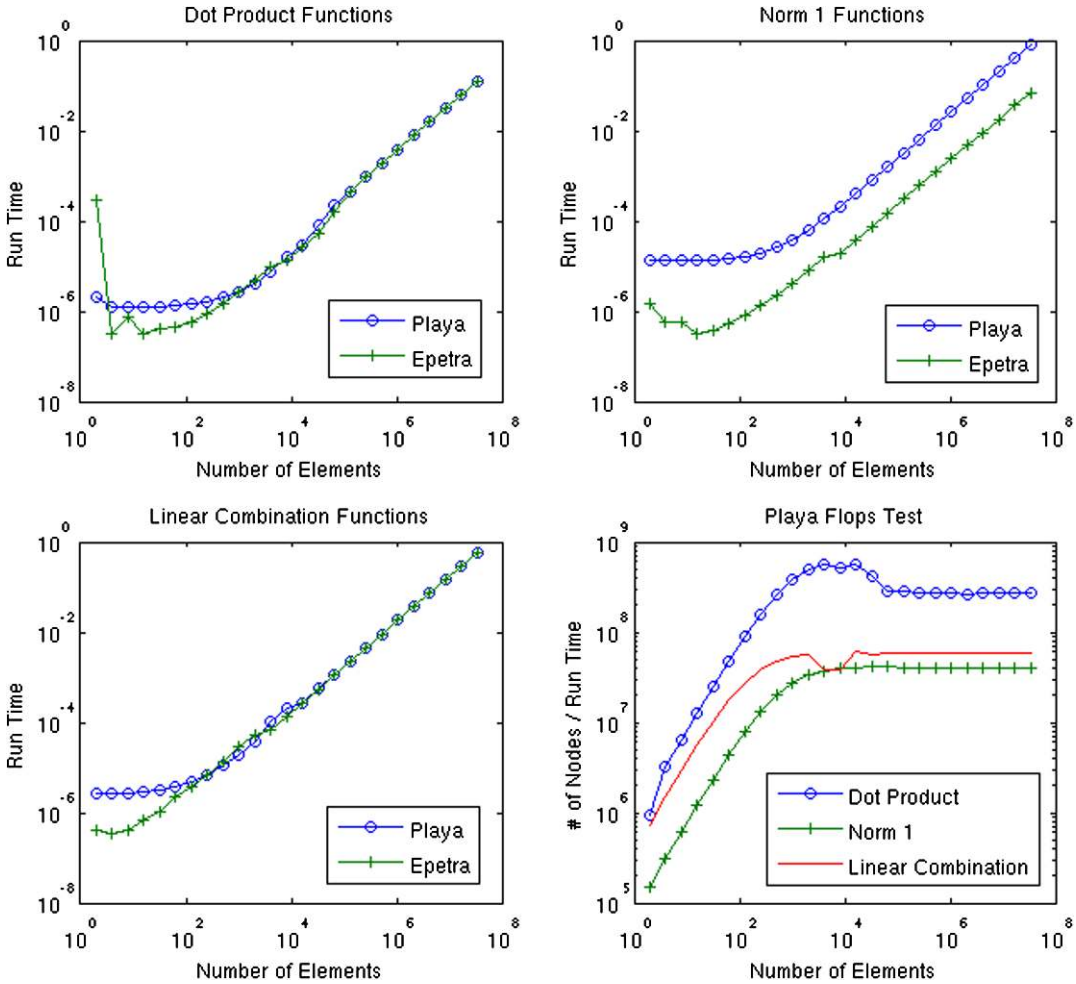


Fig. 2. Comparison of Playa and Epetra performance on platform (b). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0347.)

a streaming functor, seems to vary across platforms. In theory, the compiler should inline appropriate function calls and produce loops comparable to what Epetra contains. However, it seems that on some platforms, the compilers are not able to optimize through the several layers of templated C++ code to obtain the predicted performance. Vector operations whose performance is a critical part of overall solver performance should probably be implemented through delegation to a concrete type rather than through a streaming functor.

We also performed similar tests with two other packages, Thyra and Tpetra. We observed qualitatively similar behavior in that Thyra performance had O(1) overhead relative to Epetra but asymptotically matched the performance of Epetra. The performance of Tpetra, on the other hand, did not always match that of Epetra. We believe that a similar issue holds for Tpetra as for the vector 1-norm computation; Tpetra linear algebra is delegated to Kokkos [1], which is templated over an abstraction of computer hardware. The computations are performed via functors mapped to hardware through parallel for and reduction operations. Compiling through the templates presents a similar problem to Playa, but somewhat more pronounced as there is an additional layer of templates due to hardware. Coaxing the compilers to do the right thing on this sort of code seems to be an important open question.

5. Implementation issues

5.1. Handles and their uses

Handle objects are an essential part of the Playa design. Their most obvious use is to enable overloaded operations on runtime-polymorphic objects; however,

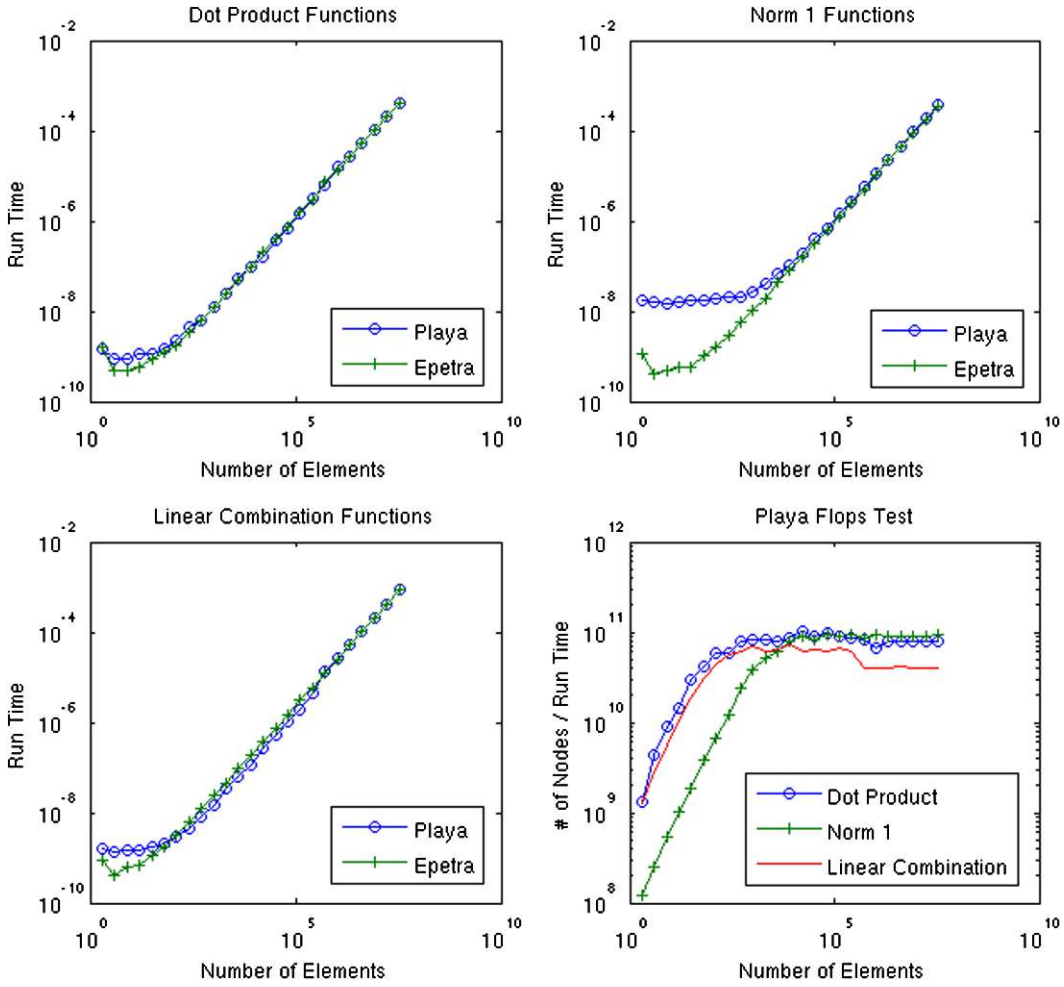


Fig. 3. Comparison of Playa and Epetra performance on platform (c). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0347.)

they also have numerous other uses. We outline just two examples: single point-of-contact error checking and safe management of object relationships.

In every operation involving two or more vectors or operators, mathematical and structural compatibility of the operands should be checked. The `isCompatible()` member function of `VectorSpace` is intended for this very purpose. There is then an issue of which object or function has the responsibility to perform these checks. The handle layer is a natural place to put such checks.

A more important use of the handle layer is to set up safe object relationships. An example of an object relationship that is simple in concept yet nontrivial to set up safely is the storage of a `VectorSpace` in a `Vector`. Recall that a vector space – an object of some `VectorSpaceBase` derived type – creates a vector via the `createMember()` function. The difficulty is that the DerivedVS instance cannot store an RCP pointing to itself (to do so would create a memory leak). The resolution is to attach the space's RCP to the vector after creation. The user can be asked to remember this step, or the logic can be wrapped in nonmember constructors. A simple alternative is to let the handle layer do the late binding, as in the following function.

```
Vector<Scalar> VectorSpace<Scalar>::createMember()
    const
{
  Vector<Scalar> rtn =
    this->ptr()->createMember(*this);
  return rtn;
}
```

This enables safe object relationships in a manner transparent to the user.

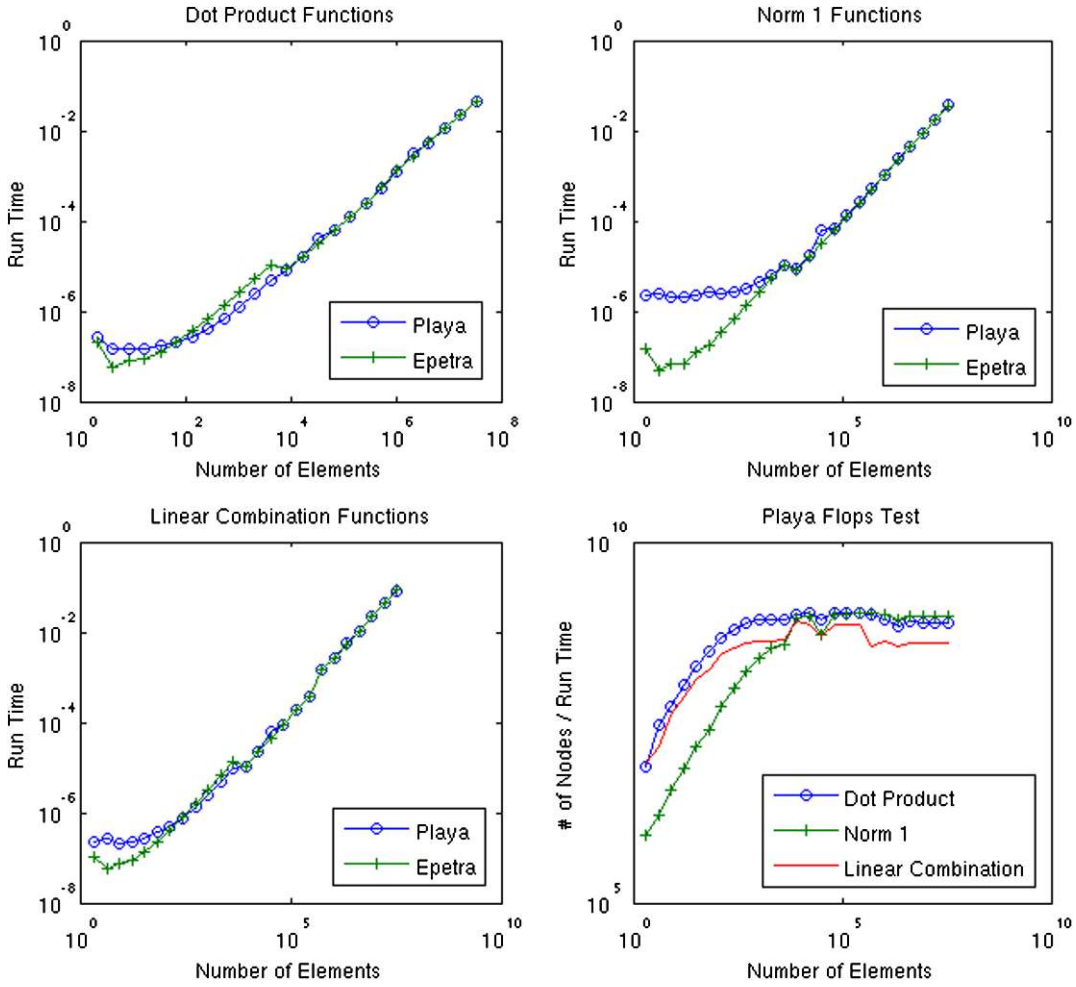


Fig. 4. Comparison of Playa and Epetra performance on platform (d). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0347.)

5.2. *Vector operations*

Augmenting an abstract vector interface to include a new vector operation is a vexing problem. In the original HCL, every vector operation required by clients was represented as a pure virtual function in the base class. TSF continued this, while providing several simplifications such as a generic traits-based adapter for concrete types. A more scalable solution was devised by Bartlett [7], who realized that if reduction and transformation operations were represented as objects, then a vector could have a single point of entry for *all* operations. The codes TSFCore and Thyra have been based on this idea; in these codes, vector operations are done exclusively via RTOp invocations.

In Playa an intermediate approach is taken: operations can be carried out either by delegation to a concrete type, or by generic RTOp. We consider it likely that developers of low-level vector libraries will do a better job optimizing their vector operations than we will, so we want to use their implementations wherever possible. Nonetheless, no library will have thought to implement every operation that might be needed by a client, so some means of scalable (in developer time) extensibility is essential. Therefore, Playa also provides a mechanism for application of generic operations via templated member functions.

5.2.1. *Functor-based vector operations*

As an example of implementing a vector operation through a templated functor, we show Playa's elementwise vector product ("dot star" in MATLAB). The `.*` operator does not exist in C++, so we use a member function `dotStar()` instead of an overloaded opera-

tor; a Playa user would write this code:

```
Vector<Scalar> z = x.dotStar(y); // z = x .* y
```

The member function is implemented in two stages: a const member function

```
template <class Scalar> inline
Vector<Scalar> Vector<Scalar>::dotStar(const
    Vector<Scalar>& other) const
{
  Vector<Scalar> rtn = space().createMember();
  rtn.acceptCopyOf(*this);
  rtn.selfDotStar(other);
  return rtn;
}
```

which calls a non-const member function that in turn calls a generic functor-streaming function with a `DotStar()` functor as an argument.

```
template <class Scalar> inline
Vector<Scalar>& Vector<Scalar>::selfDotStar(const
    Vector<Scalar>& other)
{
  return applyBinaryFunctor(PlayaFunctors::DotStar
      <Scalar>(), other);
}
```

The `DotStar()` functor object simply provides an elementwise evaluation operation,

```
template <class Scalar>
class DotStar
{
public:
  DotStar() {}

  Scalar operator()(const Scalar& x, const Scalar&
      y) const
    {return x*y;}
};
```

The `applyBinaryFunctor()` member function is templated on functor type, so the streaming logic is written once and reused for all binary functors. Similar application functions have been implemented for unary and ternary transformation functors as well as unary and binary reduction functors. Reduction functors are templated on reduction target type. Reduction functors are necessarily stateful and so must also provide callbacks for initialization and finalization.

There are several ways in which to improve the performance of the code shown above. First, the two-stage call requires two loops over the data: one for the copy in `dotStar()` (the copy is implemented as a functor application) and another for the `DotStar()` functor application in `selfDotStar()`. Second, a temporary is created. This could be avoided using deferred evaluation of an intermediate expression representation as described in the next section.

It was initially our plan to implement all of Playa's vector operations via functors. However, as discussed in Section 4, performance testing indicated that functor-based operations did not give consistently good performance across all platforms. Therefore, we implemented the critical operations for iterative solves (updates, two-norms and dot products) using delegation to the concrete type. In the future, we envision a traits mechanism for compile-time choice between functors and delegation for each concrete type.

5.3. Overloaded operations

The critical issue in efficient overloaded operations is to avoid creating temporary vectors, which would result in O(N) overhead. We solve this problem using template metaprogramming to form expression representations, followed by deferred evaluation of these expressions in a context where unnecessary overhead can be avoided. Any expression representing a linear combination of vectors is rendered at compile time into an object of class `LCN`, which is templated on the number of terms and contains statically-allocated arrays of vectors and coefficients. For example, the overloaded scalar-vector multiplication operator produces a one-term linear combination (type `LCN<Scalar,1>`),

```
template <class Scalar> inline
LCN<Scalar, 1> operator*(const Scalar& alpha,
    const Vector<Scalar>& x)
{
  return LCN<Scalar, 1>(alpha, x);
}
```

and the overloaded vector-`LCN` addition operator produces a two-term linear combination object (type `LCN<Scalar,2>`).

```
template <class Scalar> inline
LCN<Scalar, 2> operator+(const Vector<Scalar>& x,
    const LCN<Scalar, 1>& lc)
{
  const Scalar one = Teuchos::ScalarTraits<Scalar
      >::one();
  return LCN<Scalar, 2>(one, x, lc);
}
```

Thus an expression such as $\alpha x + \beta y$ is identified at compile time as a two-term linear combination, and the effect of the overloaded multiplication and addition operators is only to form an LCN object whose nodes contain the scalars and shallow-copied vectors to be operated on. No numerical calculations are done at this point. The overhead is O(1).

The expression embodied in an LCN object is only evaluated numerically upon assignment or other conversion to a vector. For example, the code for the overloaded += operator shows how a two-term linear combination formed through overloaded operations is forwarded to a low-level update() function, in which numerical computations are done.

```
template <class Scalar> inline
Vector<Scalar>& Vector<Scalar>::operator+=(const
    LCN<Scalar, 2>& lc)
{
  const Vector<Scalar>& x = lc.vec(0);
  const Scalar& alpha = lc.coeff(0);
  const Vector<Scalar>& y = lc.vec(1);
  const Scalar beta = lc.coeff(1);
  const Scalar one = Teuchos::ScalarTraits<Scalar
      >::one();

  TEUCHOS_TEST_FOR_EXCEPTION(!this->space().
      isCompatible(x.space()),
   std::runtime_error,
   "Spaces this=" << this->space() << "and other="
   << x.space() << "are not compatible in operator
       +=()");

  this->update(alpha, x, beta, y, one);

  return *this;
}
```

The next example shows the overloaded vector assignment operator. In the event that the target of the assignment is an existing vector of compatible space, we can simply overwrite its entries with the result of the evaluation, with no memory allocation needed. If the target of the assignment is null or needs reshaping, allocation is done.

```
template <class Scalar> inline
Vector<Scalar>& Vector<Scalar>::operator=(const
    LCN<Scalar, 3>& lc)
{
  const Vector<Scalar>& x = lc.vec(0);
  const Scalar& alpha = lc.coeff(0);
  const Vector<Scalar>& y = lc.vec(1);
  const Scalar& beta = lc.coeff(1);
  const Vector<Scalar>& z = lc.vec(2);
  const Scalar& gamma = lc.coeff(2);
  const Scalar zero = Teuchos::ScalarTraits<Scalar
      >::zero();

  TEUCHOS_TEST_FOR_EXCEPTION(!y.space().
      isCompatible(x.space()),
    std::runtime_error,
    "Spaces x=" << x.space() << " and y="
    << y.space() << " are not compatible in
        operator=(a*x + b*y + c*z)");

  TEUCHOS_TEST_FOR_EXCEPTION(!z.space().
      isCompatible(x.space()),
    std::runtime_error,
    "Spaces x=" << x.space() << " and z="
    << z.space() << " are not compatible in
        operator=(a*x + b*y + c*z)");

  if (this->ptr().get() != 0 && this->space() == x
      .space())
  {
    // If the LHS exists and is from the same
    // space as the RHS vectors, use the update
    // operation to compute (*this) =
    // zero*(*this) + alpha*x + beta*y + c*z
    this->update(alpha,x,beta,y, gamma,z,zero);
  }
  else
  {
    // If the vectors are from different spaces,
    // or if the LHS is null, form the RHS vector
    // and overwrite the LHS's ptr with it.
    Vector e = lc.eval();
    this->ptr() = e.ptr();
  }
  return *this;
}
```

These functions are specialized (in the sense of partial template specialization) to two-term and three-term linear combinations. Specialization to linear combinations of other sizes is a programming chore that can be done once and for all for any given linear combination size. The general case is dealt with by writing a function templated on linear combination size.

6. Conclusion

6.1. Results

We have demonstrated the possibility of a very efficient overloaded operator syntax using expression tem-

plates on polymorphic reference-counted handles to vector objects. Our experiments indicate that our overloaded operations are indistinguishable in performance from direct calls to Epetra for objects with dimension greater than 100, as shown in the comparisons. Playa's compact representation of block-structured operators, implicit inverses and transposes, and operator overloading has enabled the easy development of block preconditioners for problems such as incompressible flow and thermal fluids. Importantly, Playa builds on the successes of many existing Trilinos packages without replicating functionality, increasing their usability without impeding the robustness or performance. This suggests a real possibility for Playa maturing into a development environment for production-ready high-level algorithms.

6.2. Future directions

Yet, arriving at such a position will require some additional development. For one, our functor-based approach for computing on data chunks does not perform reliably across platforms. This will be an important issue to resolve. Second, there are some minor inefficiencies internal to some of our implicit classes that careful profiling and optimization will improve (e.g., compound operators for temporary vectors at each application). A third opportunity will be to include more vector, matrix, and solver tools within the Playa interface (e.g., Tpetra, PETSc). Doing so should also enable greater portability to emerging multicore-based architectures, including GPU and hybrid platforms. Finally, ongoing research aims to expand Playa's capabilities by developing tools such as nonlinear solvers and optimization algorithms entirely within the Playa framework.

References

[1] C. Baker, M. Heroux, H. Edwards and A. Williams, A light-weight API for portable multicore programming, in: *Proc. 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, IEEE, Washington, DC, 2010, pp. 601–606.

[2] W. Bangerth, R. Hartmann and G. Kanschat, deal.II – a general-purpose object-oriented finite element library, *ACM Trans. Math. Softw.* **33** (2007), 24/1–24/27.

[3] R. Bartlett, Teuchos::RCP beginner's guide: an introduction to the Trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++, Technical Report SAND2004-3268, Sandia National Laboratories, 2004.

[4] R. Bartlett, Thyra coding and documentation guidelines (tcdg) version 1.0, Technical Report SAND2010-2051, Albuquerque, NM and Livermore, CA, 2010.

[5] R. Bartlett, M. Heroux and K. Long, TSFCore: a package of light-weight object-oriented abstractions for the development of abstract numerical algorithms and interfacing to linear algebra libraries and applications, Technical Report SAND2003-1378, Sandia National Laboratories, 2003.

[6] R.A. Bartlett, Thyra: Interfaces for abstract numerical algorithms, available at: http://trilinos.sandia.gov/packages/thyra/, 2008.

[7] R.A. Bartlett, B.G.V.B. Waanders and M.A. Heroux, Vector reduction/transformation operators, *ACM Trans. Math. Softw.* **30** (2004), 62–85.

[8] H.C. Elman, V.E. Howle, J. Shadid, R. Shuttleworth and R. Tuminaro, Block preconditioners based on approximate commutators, *SIAM J. Sci. Comput.* **27** (2006), 1651–1668.

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[10] M.S. Gockenbach, M.J. Petro and W.W. Symes, C++ classes for linking optimization with complex simulations, *ACM Trans. Math. Soft.* **25** (1999), 191–212.

[11] V.E. Howle and R.C. Kirby, Block preconditioners for finite element discretization of incompressible flow with thermal convection, *Numer. Linear Algebra Appl.* **19** (2012), 427–440.

[12] D. Kay, D. Loghin and A. Wathen, A preconditioner for the steady-state Navier–Stokes equations, *SIAM J. Sci. Comput.* **24** (2002), 237–256.

[13] R. Kirby, A new look at expression templates for matrix computation, *Comput. Sci. Eng.* **5** (2003), 66–70.

[14] T.G. Kolda and R.P. Pawlowski, NOX nonlinear solver project, available at: http://software.sandia.gov/nox.

[15] A. Logg and G. Wells, Dolfin: automated finite element computing, *ACM Trans. Math. Softw.* **37** (2010), 1–28.

[16] K. Long, R. Bartlett, P. Boggs, M. Heroux, P. Howard, V. Howle and J. Reese, TSFExtended user's guide, Technical report, Sandia National Laboratories, 2004.

[17] K. Long, R. Kirby and B. van Bloemen Waanders, Unified embedded parallel finite element computations via software-based Fréchet differentiation, *SIAM J. Sci. Comput.* **32** (2010), 3323–3351.

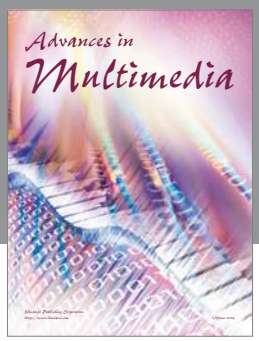
[18] S.D. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, Reading, MA, 2005.

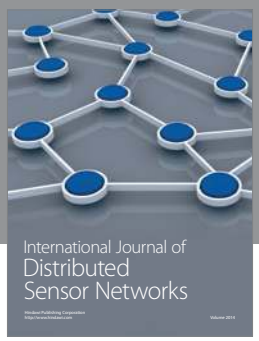
[19] J.N. Shadid, E.C. Cyr, R.P. Pawlowski, R.S. Tuminaro, L. Chacón and P.T. Lin, Initial performance of fully-coupled AMG and approximate block factorization preconditioners for solution of implicit FE resistive MHD, in: *Proc. 5th European Conference on Computational Fluid Dynamics*, Lisbon, Portugal, 2010.

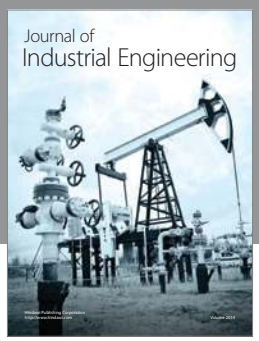
[20] L.N. Trefethen and I. David Bau, *Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.

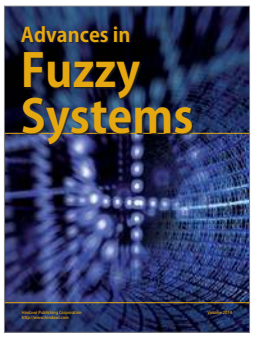
[21] T. Veldhuizen, Expression templates, *C++ Report* **7** (1995), 26–31.

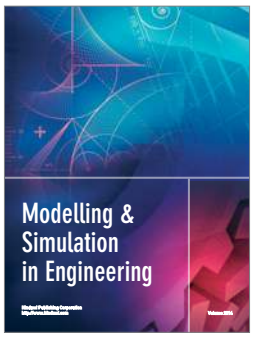
[22] T. Veldhuizen, Arrays in blitz++, in: *Proc. 2nd Int. Symp. on Computing in Object-Oriented Parallel Environments*, Springer, London, UK, 1998, pp. 223–230.

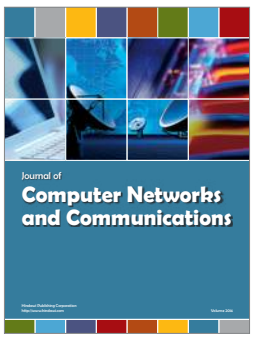












Hindawi
Submit your manuscripts at
http://www.hindawi.com