Generic Matrix Multiplication and Memory Management in LinBox

Erich Kaltofen
North Carolina State University
kaltofen@math.ncsu.edu

Dmitriy Morozov
Duke University
morozov@cs.duke.edu

George Yuhasz
North Carolina State University
gyuhasz@math.ncsu.edu

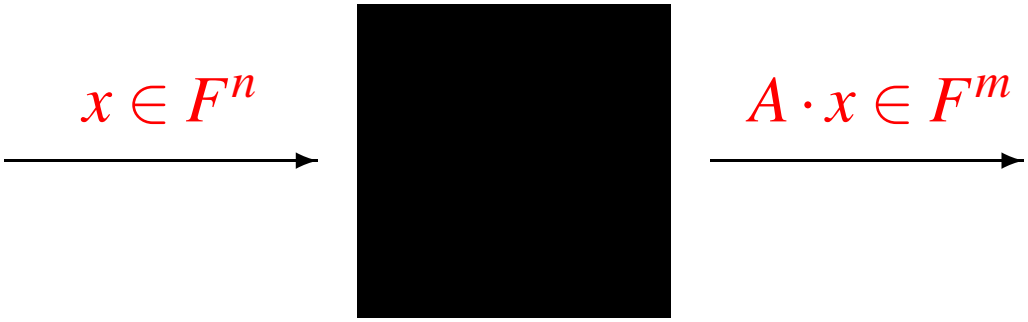
LinBox

C++ template library of black box linear algebra algorithms

LinBox

C++ template library of black box linear algebra algorithms

- black box linear algebra

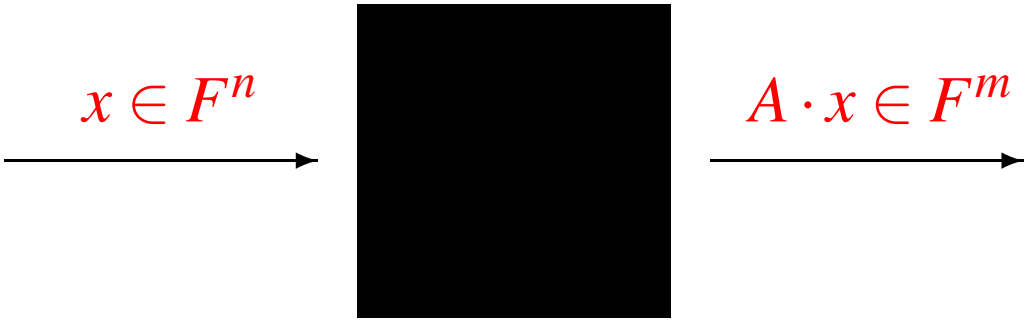


F an arbitrary field

LinBox

C++ template library of black box linear algebra algorithms

- black box linear algebra



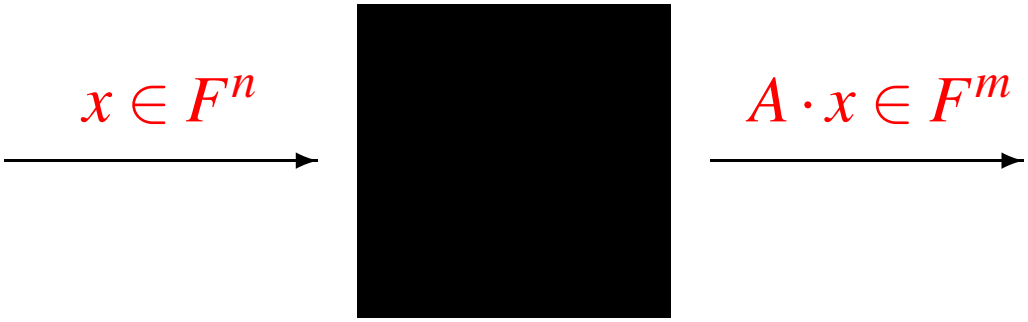
F an arbitrary field

- generic, reusable software design
  - common object interfaces
  - external libraries for field arithmetic

LinBox

C++ template library of black box linear algebra algorithms

- black box linear algebra

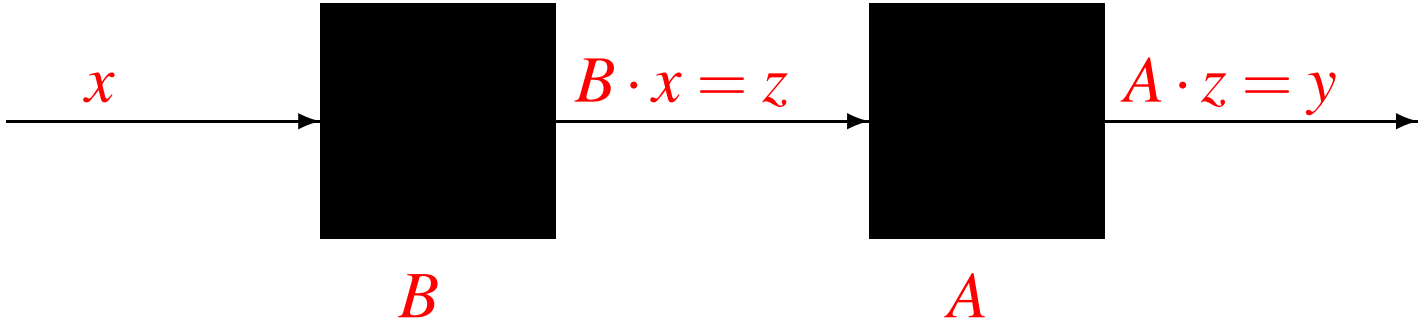


F an arbitrary field

- generic, reusable software design
  - common object interfaces
  - external libraries for field arithmetic
- `www.linalg.org` for release 1.0

Overview

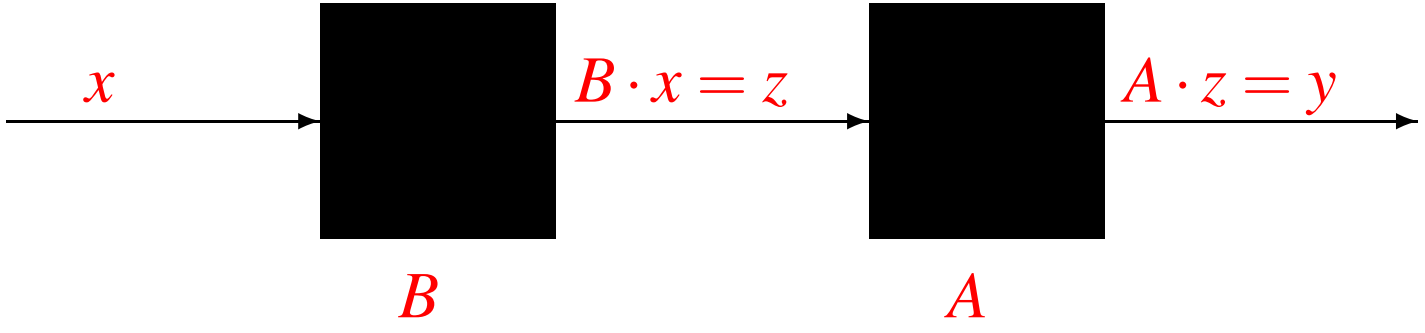
- black box matrix multiplication
  - lazy evaluation scheme
    compute $y = AB \cdot x$



  - vector type of z ??

Overview

- black box matrix multiplication
  - lazy evaluation scheme
    compute $y = AB \cdot x$



  - vector type of z ??
- external memory management
  - incorporation of various memory models
  - example using garbage collected library SACLIB

Black Boxes

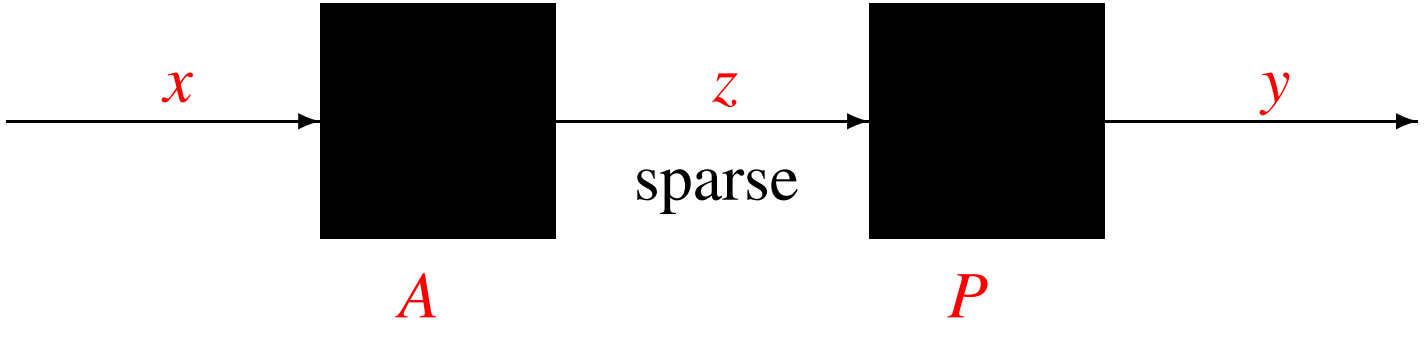
- common object interface of matrices
- field and dimension methods
- member templates `apply` and `applyTranspose`
  - template parameters `Input` and `Output`
- `PreferredInputVector` and `PreferredOutputVector`

Choices for the Intermediate Vector Type

Matrix	Preferred Input	Preferred Output
A	dense	sparse
P	sparse	sparse

Choices for the Intermediate Vector Type

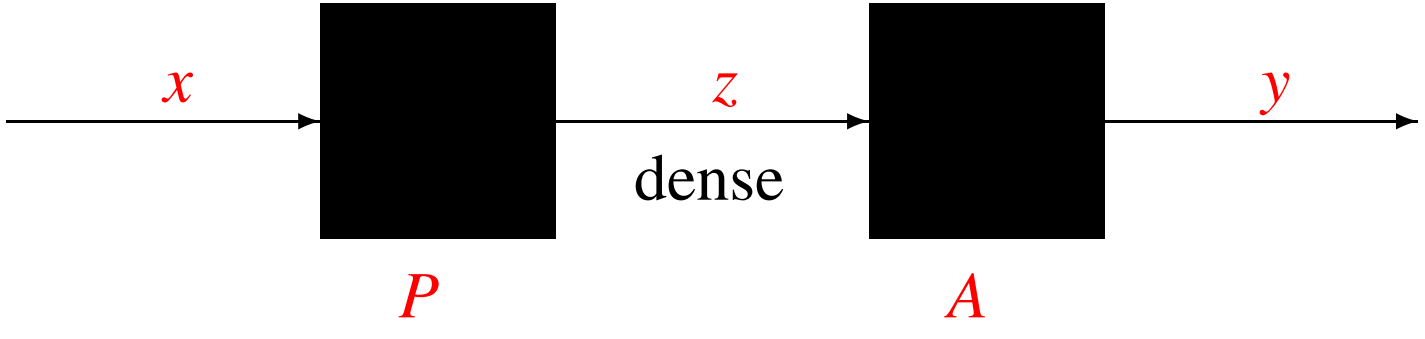
Matrix	Preferred Input	Preferred Output
A	dense	sparse
P	sparse	sparse



vector types equal

Choices for the Intermediate Vector Type

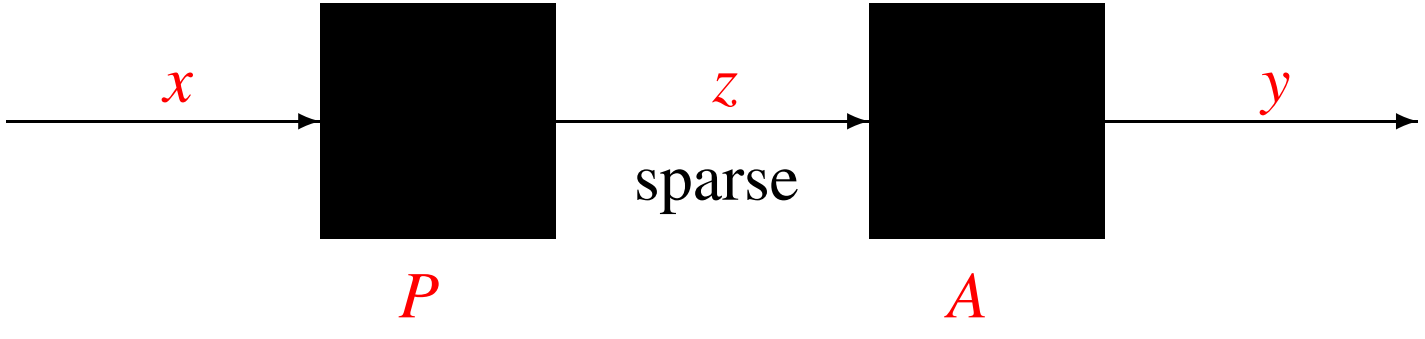
Matrix	Preferred Input	Preferred Output
A	dense	sparse
P	sparse	sparse



increase efficiency

Choices for the Intermediate Vector Type

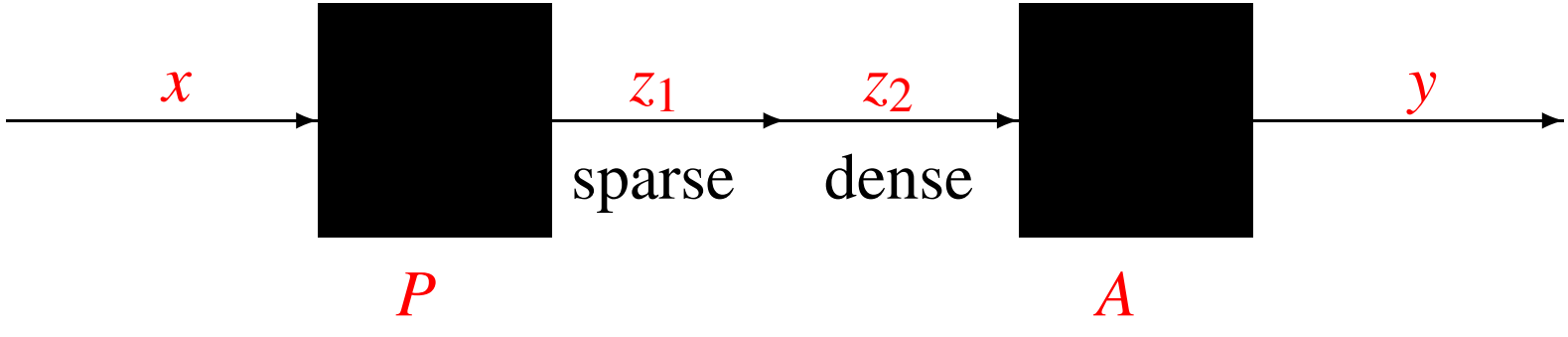
Matrix	Preferred Input	Preferred Output
A	dense	sparse
P	sparse	sparse



conserve memory

Choices for the Intermediate Vector Type

Matrix	Preferred Input	Preferred Output
A	dense	sparse
P	sparse	sparse



maximize efficiency

Template Class `Compose`

- interface of black box matrix multiplication

Template Class `Compose`

- interface of black box matrix multiplication

```
template <class _Blackbox1,
          class _Blackbox2=_Blackbox1,
          IntermediateVector flag=DEFAULT >
class Compose;
```

Template Class `Compose`

- interface of black box matrix multiplication
- single instance of the intermediate vector(s)

Template Class `Compose`

- interface of black box matrix multiplication
- single instance of the intermediate vector(s)
- four methods of selecting the intermediate vector type
  - INPUT – uses the preferred input type of the left matrix
  - OUTPUT – uses the preferred output type of the right matrix
  - DEFAULT – uses dense vector if types are unequal, otherwise the shared preferred type
  - CONVERSION – uses both preferred types and converts between the two

Template Class `Compose`

- interface of black box matrix multiplication
- single instance of the intermediate vector(s)
- four methods of selecting the intermediate vector type
  - INPUT – uses the preferred input type of the left matrix
  - OUTPUT – uses the preferred output type of the right matrix
  - DEFAULT – uses dense vector if types are unequal, otherwise the shared preferred type
  - CONVERSION – uses both preferred types and converts between the two
- specialized for each option by `flag` template parameter

Implementation of `Compose`

- template meta programming (template if-then-else controls)

Implementation of `Compose`

- template meta programming (template if-then-else controls)
- DEFAULT specialization
  - intermediate types compared and a type selection made
  - comparison and selection made at compile time
  - partially specialized type choosing class

Implementation of `Compose`

- template meta programming (template if-then-else controls)
- DEFAULT specialization
  - intermediate types compared and a type selection made
  - comparison and selection made at compile time
  - partially specialized type choosing class
- CONVERSION specialization
  - intermediate types compared to avoid unnecessary copying
  - partially specialized nested class

Memory Managed External Libraries

- common object interface for memory management
- pointer and reference types
- `new` and `delete`
- generic and STL compatible

SACLIB

- library of C programs derived from the SAC2 system
- modular arithmetic algorithms
- objects manipulated via handles called `Words`
- garbage collection
- allocation of garbage collected arrays
- allocating objects on the stack result in program failure

Interfacing LinBox and SACLIB

- objects stored in arrays
- arrays allocated by SACLIB function calls
- reference to the array placed in a global list
- list stores references to all SACLIB objects in use
- list registered with the garbage collector

Memory Allocation in LinBox

- field arithmetic encapsulated in a common object interface
- field element representation hidden in field implementations
- only fields operate on field elements
- black boxes, vectors and algorithms

Allocators

- generic memory management not just a LinBox problem
- Stephen Watt (ICMS 2002)
- Richardson and Krandick (CASC 2005)
- C++ Standard Template Library
- allocators
- common object interface for memory management
- STL defines a default allocator

STL Allocator Interface

```
template <class T> class std::allocator
{
  public:
     typedef T           value_type;
     typedef size_t      size_type;
     typedef ptrdiff_t   difference_type;
     typedef T*          pointer;
     typedef const T*    const_pointer;
     typedef T&          reference;
     typedef const T&    const_reference;
```

STL Allocator Interface

```
template <class T> class std::allocator
{
  public:
     pointer address(reference r) const
        { return &r; }
     const_pointer address(const_reference r)
        const { return &r; }
     allocator() throw();
     template <class U>
        allocator(const allocator<U>&) throw();
    ~allocator() throw();
     template <class U> struct rebind
        { typedef allocator<U> other; }
```

STL Allocator Interface

```
template <class T> class std::allocator
{
  public:
     // space for n Ts
     pointer allocate(
        size_type n,
        allocator<void>::const_pointer hint = 0);
     // deallocate n Ts, don't destroy
     void deallocate(pointer p, size_type n);
     // initialize *p by val
     void construct(pointer p, const T& val)
        { new(p) T(val); }
     // destroy *p but don't deallocate
     void destroy(pointer p) { p-> T(); }
     size_type max_size() const throw();
     };// end allocator
```

Allocators and the C++ Standard

- allocator restrictions in the C++ Standard

Allocators and the C++ Standard

- allocator restrictions in the C++ Standard

Implementations of containers described in this International Standard are **permitted to assume** that their Allocator template parameter meets the following two additional **requirements** beyond those in Table 32:

— The typedef members `pointer`, `const_ pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.

Allocators and the C++ Standard

- allocator restrictions in the C++ Standard

Implementors are **encouraged** to supply libraries that can accept allocators that encapsulate more **general memory models** and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

Allocators and the C++ Standard

- allocator restrictions in the C++ Standard
- most compilers assume the requirements
  - general memory models such as distributed memory are impossible
- problem brought to the attention of Matt Austern
  - `http://lafstern.org/matt/wishlist.html`

Allocators in LinBox

- field arithmetic object interface includes allocators
  - a typedef member `ElementAllocator`
  - a method
    `ElementAllocator getElementAllocator() const`

Allocators in LinBox

- field arithmetic object interface includes allocators
  - a typedef member `ElementAllocator`
  - a method `ElementAllocator getElementAllocator() const`
- `ElementAllocator` used in STL containers
  - vectors in LinBox are STL vectors, lists and maps
  - black boxes use STL containers to store field elements

Allocating Temporary Elements

- call allocate directly
  - possible memory leaks if deallocate is forgotten

Allocating Temporary Elements

- call allocate directly
  - possible memory leaks if deallocate is forgotten
- "resource acquisition is initialization"

```
some_algorithm()
{ Field f;
  ...
  std::vector<Field::Element, Field::ElementAllocator>
  tmp_vec(2, Field::Element(), f.getElementAllocator());
  Field::ElementAllocator::reference one = tmp_vec[0];
  Field::ElementAllocator::reference two = tmp_vec[1];
  ...}
```

Using SACLIB in LinBox

LinBox needed the following to access SACLIB's functionality

- a field interface `SacLibModularField`
- a class `SacLibAllocator`
- initialization in an unnamed scope

Thank you

Visit www.linalg.org for more information on LinBox and release 1.0