Novel algebras for advanced analytics in Julia

Novel Algebras for Advanced Analytics in Julia

Viral B. Shah*, Alan Edelman†, Stefan Karpinski‡, Jeff Bezanson§, Jeremy Kepner¶
*Email: viral@mayin.org
†Email: edelman@math.mit.edu
‡Email: stefan@karpinski.org
§Email: bezanson@mit.edu
¶Email: kepner@ll.mit.edu

***Abstract*—A linear algebraic approach to graph algorithms that exploits the sparse adjacency matrix representation of graphs can provide a variety of benefits. These benefits include syntactic simplicity, easier implementation, and higher performance. One way to employ linear algebra techniques for graph algorithms is to use a broader definition of matrix and vector multiplication. We demonstrate through the use of the Julia language system how easy it is to explore semirings using linear algebraic methodologies.**

I. Introduction

A. Semiring algebra

The duality between the canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation has been a part of graph theory since its inception [5], [6]. Matrix algebra has been recognized as a useful tool in graph theory for nearly as long (see [3] and references therein). A linear algebraic approach to graph algorithms that exploits the sparse adjacency matrix representation of graphs can provide a variety of benefits. These benefits include syntactic simplicity, easier implementation, and higher performance. One way to employ linear algebra techniques for graph algorithms is to use a broader definition of matrix and vector multiplication. One such broader definition is that of a semiring. For example, in semiring notation we write matrix-matrix multiply as:

$$C = A +.* B$$

where

$$+.*$$

denotes standard matrix multiply. In such notation, a semiring requires that addition and multiplication are associative, addition is commutative, and multiplication distributes over addition from both left and right. In graph algorithms, a fundamental operation is matrix-matrix multiply where both matrices are sparse. This operation represents multi source 1-hop breadth first search (BFS) and combine, which is the foundation of many graph algorithms [2]. In addition, it is often the case that operations other than standard matrix multiply are desired, for example:

1) MaxPlus: $C = A \max.+ B$
2) MinMax: $C = A \min.\max B$
3) OrAnd: $C = A |.\& B$
4) General (f and g) : $C = A f.g B$

With this more general case of sparse matrix multiply, a wide range of graphs algorithms can be implemented [4].

II. Application example

A classic example of the utility of the semiring approach is in finding the minimum path between all vertices in graph (see [11] in [4]. Given a weighted adjacency matrix for a graph where $A(i,j) = w_{ij}$ is the weight of a directed edge from vertex i to vertex j. Let $C(i,j)_2$ be the minimum 2-hop cost from vertex i to vertex j. C_2 can be computed via the semiring matrix product:

$$C_2 = A \ \min.+ A$$

Likewise, C_3 can be computed via

$$C_3 = A \ \min.+ A \min.+ A$$

and more generally

$$C_k = A^k$$

III. Julia

It has become clear that programmers and scientists prefer high-level, interactive, dynamic environments for algorithm development and data analysis. Systems such as Matlab [7], Octave [9], R [10], SciPy [8], and SciLab [12] provide greatly increased convenience and productivity, yet C and Fortran remain the gold standard for computationally-intensive problems because these high-level dynamic systems still lag significantly in performance. As a result, the most challenging areas of technical computing have benefited the least from the increased abstraction and productivity offered by higher level languages. Julia [1] is a high-level, dynamic language, designed from the ground up to take advantage of modern techniques for executing dynamic languages efficiently. As a result, Julia has the performance of a statically compiled language while providing the interactive, dynamic experience and productivity that scientists have come to expect.

Julia also introduces many powerful computer science tools to scientific computing, including a sophisticated type system, multiple dispatch, coroutines, Lisp-style metaprogramming (including real macros), and built-in support for distributed computation. Although a powerful type system is made available to the programmer, it remains unobtrusive in the sense that one is never required to specify types, nor is performance dependent upon doing so: unless the programmer wants to take advantage of Julia's dispatch system or create C-compatible

types, their code will work just as well without ever mentioning types. Likewise, coroutines, macros and distributed programming primitives are right there, should the programmer ever need them, but are not required for day-to-day programming.

We recommend trying the code examples in this paper. Just a few moments may convince the reader that Julia is simple yet powerful. All of the code may be found on GitHub: `github.com/ViralBShah/SemiringAlgebra.jl` . First time users should readily be able to google, download, and install Julia.

Julia is a novel new paradigm constructed with high performance computing in mind. Roughly speaking, high performance computing has been significantly difficult that getting codes up and working and in production has taken so much time, that there all too often has been little time for algorithmic exploration or software experimentation. Julia is a game changer for high performance computing; but that is not the focus of this work. In this work, our goal is to demonstrate the expressiveness and utility of Julia in the context of semirings. We invite readers to imagine what a semiring implementation might look like, how long it would take to implement, and how it might perform using their favorite programming methodology.

IV. Semiring algebra in Julia

Julia's ordinary matrix multiplication is recognizable to users of many high level languages:

```
A=rand(m,n)
B=rand(n,p)
C=A*B
```

or one can equally well use the prefix notation

```
C = *(A,B)
```

instead of the infix notation.

We believe that users of matrix-multiply who are used to such compact expressions would prefer not to have overly complicated syntax to express the more general semirings. Julia offers two approaches that are readily available for exploration:

1) Star overloading: This method is recommended for interactive exploration of many semiring operators. Without introducing any types, define `*(f,g)(A,B)` to perform the semiring operation very generally, with no a-priori restriction on the binary functions `f` or `g`. Upon overloading the star operator, and defining the ringmatmul, the following immediately work in Julia:

```
*(max,+)(A,B)
*(min,max)(A,B)
*(|,&)(A,B)
*(+,*)(A,B)
```

The last example computes the usual matrix product.

2) Creation of semiring objects: This method is recommended for users who are working exclusively in one semiring and wish to optimize notation and performance. In this method, a semiring type is created, and one overloads scalar + and scalar * only. Julia's generic definitions for matrix multiplication, which depends only on having appropriate definitions for + and * does the rest, allowing one to immediately compute matrix products in the newly defined semiring, using the usual notation for matrix products.

```
function ringmatmul(+, *,
                    A::Matrix,B::Matrix)
  m, p = size(A); n = size(B,2)
  C = [A[i,1]*B[1,j] for i=1:m, j=1:n]
  for i=1:m, j=1:n, k=2:p
    C[i,j] += A[i,k]*B[k,j]
  end
  return C
end

*((+)::Function, (*)::Function) =
              (A,B)->ringmatmul(+,*,A,B)
```

Fig. 1. SemiRing matrix multiply by overloading + and *. This seemingly textbook matrix multiply routine is anything but ordinary because + and * are local variables. Inside the subroutine + and * take on any semiring operations with which they are called. Thanks to Julia internals, users may execute this code at the prompty by simply typing `*(max,+)(A,B)` or the more familiar infix notation `A *(max,+)B` .
Overloading allows users to conveniently explore multiple semiring operations on the same data.

A. Star Overloading

The star overloading functionality may be explored by typing the example in Figure IV-A directly into a Julia session. The `ringmatmul` function is straightforward – it is a standard naïve triple loop matrix multiply. The function takes four arguments, the first two, + and *, are functions that will serves as the local versions of addition and multiplication, while the last two are the matrices to multiply using those operations. This highlights a few relevant features of the Julia language. First, functions are first-class values that can be passed into other functions as arguments and then used with standard function call syntax. Second, the + and * operators are just regular functions with some special syntax. For example, the expression `C[i,j] += A[i,k]*B[k,j]` is translated into `C[i,j] = +(C[i,j],*(A[i,k],B[k,j]))` where the functions + and * are looked up in the current scope just like any other variables would be. The second function definition says that if "*" is called with two arguments, both of which have the type `Function`, then the result is a function itself which takes two arrays as arguments, and calls `ringmatmul` with arguments `f,g,A,B`. This highlights a few more features of Julia: it uses multiple dispatch by proding a new behavior for * when its areguments are functions; it returns a function as a value, which can then be used elsewhere, again with the standard function call syntax.

B. Semiring Objects

Here we create a "max-plus" semiring type. Other semirings can be implemented by changing the definitions of "+" and "*", which are set to use "max" and "plus" (Figure IV-B). The new type does not need to be taught matrix multiply; matmul will work with any underlying numeric type (e.g. int, float, ...), and will also work with both dense and sparse matrices. Arrays of the new semiring type will use the same

```
immutable MPNumber{T} <: Number
    val::T
end

+(a::MPNumber, b::MPNumber)
      = MPNumber(max(a.val, b.val))
*(a::MPNumber, b::MPNumber)
      = MPNumber(a.val+b.val)
show(io::IO, k::MPNumber)
      = print(io, k.val)

zero{T}(::Type{MPNumber{T}})
      = MPNumber(typemin(T))
one{T}(::Type{MPNumber{T}})
      = MPNumber(zero(T))
promote_rule{T<:Number}(::Type{MPNumber},
                        ::Type{T})
      = MPNumber

mparray(A::Array) = map(MPNumber, A)
array{T}(A::Array{MPNumber{T}})
      = map(x->x.val, A)
mpsparse(S::SparseMatrixCSC)
      = SparseMatrixCSC(S.m, S.n, S.colptr,
            S.rowval, mparray(S.nzval))
```

Fig. 2. An altnerative approach to that of Figure 1 is the MPNumber type with Max-Plus Algebra properties. Comparing with the overloading approach, this approach automatically works with dense and sparse matrices of any type without requiring the user to redefine matrix multiply. The code sets up a MaxPlus number (`MPNumber`), defines a plus and times operator `max` and `+`, sets ut the identity elements (`zero=typeminT` and `one=zero(T)`). Users simply type `A*B` on arrays of the right type, and the semiring operation will just work. (See Fig. 3)

amount of memory as "primitive" arrays of the underlying numeric type.

The first definition describes the `MPNumber` type. Then plus and times are defined as max and plus respectively, followed by a routine IO function.

The zero and one in this semiring are defined as the identity elements for max and plus, and the promote rule is a Julia construction which allows semiring numbers and ordinary numbers to work together.

Finally the last three functions allow for the conversion between ordinary arrays and the semiring, with a special extra constructor for the important use case of sparse arrays.

Some examples of using the code are presented in the figure below:

V. Performance

Pne. The goal of Julia is to have the right combination of reasonable performance for the machine and productivity for the human. On a Macbook Pro, with dual-core 2.4 GHz Intel core i5, 8GB RAM, we compared a dense BLAS run of matmul with an MPNumber{Float64} run in the semiring. The BLAS time was 0.26msec (practically free!) for a 100×100 array. The semiring time was 313 msec. This is no surprise.

```
# Random MPNumber array
A=mparray(rand(3,3))
B=mparray(rand(3,3))

# Multiply two matrices
A*B

# Square a matrix
A^2

# Create a sparse identity matrix
C=mpsparse(sparse(eye(3,3)))

# Square a sparse matrix
C*C
```

Fig. 3. Example creation of an `mparray`. Users simply create an mparray, and operate normally from there. The operations $\star$ and $\wedge$ are automatically the max-plus operations without any user definition.

More interesting and more useful is the comparison for sparse matrices. For an $n \times n$ sparse random array with density $1/n$, where $n = 100,000$ the ordinary matrix multiply took 21 msec, and the semiring matrix multiply took a comparable 22 msec. For a higher density of $5/n$, we found that the ordinary sparse matrix multiplication again was on par with the semiring version, taking 450 msec in each case.

VI. Conclusion

Julia facilitates the implementation and exploration of graph algorithms through the semiring notation of generalized matrix mutliply. These algorithms appear in applications to data analysis and related fields. Exploitation of sparse data strucutres provides easy implementations with reasonable performance. Sparse semiring performance is comparable to ordinary sparse matmul in the Julia language.

VII. Acknowledgements

We gratefully acknowledge funding from Citigroup, Intel ISTC, VMware, and the Deshpande Foundation. The other authors thank Jeremy Kepner for raising our awareness of the importance of semirings in high performance computing and data analysis.

REFERENCES

[1] Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. http://arxiv.org/abs/1209.5145, 2012.

[2] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Sciences and Engineering*, 10(2):20–25, Mar/Apr 2008.

[3] Frank Harary. *Graph Theory*. Addison-Wesley Publishing, 1969.

[4] Jeremy V. Kepner and J. R. Gilbert. *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[5] Dénes Kőnig. Gráfok és mátrixok. *Matematikaiés Fizikai Lapok*, 38:116–119, 1931.

[6] Dénes Kőnig. *Theorie der Endlichen und Unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe*. Akad. Verlag., 1936.

[7] MATLAB. http://www.mathworks.com.

[8] Numpy. http://www.numpy.org.

[9] Octave. http://www.octave.org.

[10] R. http://www.r-project.org.

[11] Charles M. Rader. Connected components and minimum paths. In Jeremy V. Kepner and J. R. Gilbert, editors, *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[12] Scilab. http://www.scilab.org.