Course 18.337 Final Project Report: D4M in Julia

Alexander Y Chen, Dr. Jeremy Kepner, Prof. Alan Edelman

November 21 2015

Abstract

In this project, Dynamic Distributed Dimensional Data Model (D4M) was implemented in Julia. D4M is a database computation system made in MIT Lincoln Lab originally coded in Octave and Matlab. D4M formulates an associative array that includes string row and column indexing and edge values, unlike the logical Associative Collection that was already implemented in Julia. This project aims to enhance Julia's capability in database computation by adding a D4M module. This report summarizes the development and contribution thus far with discussion on design points during the importing process from Matlab.

Introduction

In essence, D4M is an "interface to support efficient development of mathematically based analytics", (Kepner et. al). D4M allows four commonly uses categories of data storages: Associative Array, Tuple Store, Parallel Client, Distributed Array, to be represented in one common mathematical representation. Thus, it allows intuitive development of practical programs to mathematically operate on these storages through matrix operations. By building on top of Julia, D4M enables Julia to compute String triplet store data intuitively.

Goal

The goal of this project is to implement a base level of D4M in the Julia way, and push its performance as close to Julia's native operations as possible. There are two main bottleneck for D4M: string sorting and sparse matrix operation. This project can be considered be successful only if :

•

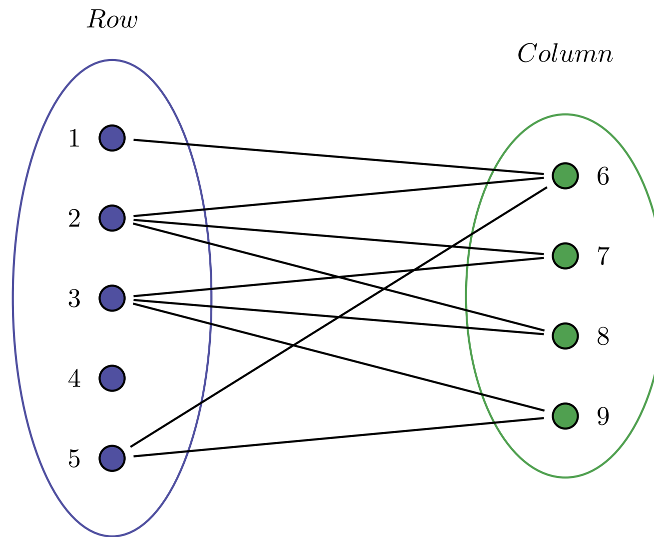


Figure 1: Data captured by Dictionary in Julia

	6	7	8	9
1	edge1			
2	edge2	edge3	edge4	
3		edge5	edge6	edge7
4				
5	edge8			edge9

Figure 2: Data captured by D4M in Julia

Implementation

There are some shared properties between Matlab and Julia, and will drive the implementation of D4M in Julia. Since both are computational languages, they chiefly work on numerical structures and values. Thus, instead of writing an object oriented representation of D4M that overloads the native Julia sparse matrix to handle D4M's string values and operations, a wrapper is made to augment the native sparse matrix. This allows the native string and number interaction operation to be mostly intact when D4M is loaded. (Ex. the sum and multiplication between a number and string shouldn't be defined) Though the possibilities of overloading mathematical operations to enable fluid interaction between number and string data could be interesting, this will hold for further studies after the core importation.

There are three driving factors in this implementation.

1. time
2. foundation
3. performance

First, the implementation has to be completed in a semester; simply put the project has to be completed for the project. Secondly, the implementation has to form a foundation, enough for me or future researchers to build upon this implementation. Thirdly, the implementation has to be Julia. In this sense, the implementation shouldn't be a direct port from another language, but should utilize and base off of Julia's core style and functions.

From this point onwards, associative array will be exclusively referencing the associative array in D4M, which includes edge information between the two associated subsets.

Memory Structure

The associative array in D4M is implemented by building on top of the sparse matrix. There were two potential styles of implementation: overloading the sparse matrix into taking String agruments, or building a String wrapper that maps sparse matrix numberic values. Julia does support overloading basic functions and memory structures quite easily. However, to enable such overloading, basic numeric and string joint operations would need to be overrided. Such basic overloading could bleed into other packages and impact their functions.

Thus instead, the less intrusive implemenation is chosen.

Key Helper Functions

There are a few key sub function that is isolated from the operations of D4M. They were not in the original implementation of D4M, but is deemed important such that it should be isolated out. The isolation would allow future improvement to augment D4M for clusters or other features

- Sorted Intersect : Calculate the intersect of two arrays with unique and sorted elements.
- Sorted Union : Calculate the union of two arrays with unique and sorted elements.
- Search Sorted Mapping : Calculate mapping from the first input sorted array to the second input sorted array
- Condense : Eliminate empty rows and columns.
- Deep Condense : Eliminate empty rows, columns, and value mapping.

Sorted Intersect. This came to a surprise that this doesn't exist in the Julia general. Just like sortedsearch give performance benefit for searching on a sorted array, sortedintersect does the same for intersect(X,Y). Set operations which would gain performance boost on sorted arrays or collection like sorted intersect could be an useful addition for Juila.

Sorted Union. This is similar to Sorted Intersect, it would iterate through two arrays with sorted and unique elements. Unlike Sorted Intersect, Sorted Union would traverse through both arrays, thus its performance dependent on the longer of the pair of arrays instead of the shorter.

Search Sorted Mapping. This is to map between two arrays, which the first sorted array is assume be a subset of the second sorted array. This helped after the Union and Intersect is found and the mapping between the new assoc to the older assoc needed to be found.

Condensing is the garbage collecting feature for D4M. They eliminate empty row, column, and values mapped with D4M. These functions are isolated out for benefit of future functions. Though some functions would benefit from doing these gabarge collecting during the operation, having isolate functions to call to manually improve data or for functions that can't

Condense. This is a critical function in keeping the sparse matrix size managable. It would eliminate empty rows and columns in the sparse matrix. Though currently only the subreferencing (indexing) and multiply utilizes condense. It is a simple enough function, but could be utilized by future functions that is more complex.

Deep Condense. This is to manage the value mapping just as condense do for row and column mapping. However, it does call condense and clear up the row and column, so essentially this will remove all empty row, column, and unused values.

Examples

There are three main folders of examples that are implemented.

These examples were implemented in Matlab for a class on the original D4M. Their Julia counterpart is implemented in the D4M module.

Assoc Intro

Assoc Intro is a collection of example meant to demonstrate operations of Assoc. This includes construction, indexing, and mathematical operations.

1. 01Setup : This setups for the introduction and demo the basic constructor and write out.

2. 02subref : This showcases the equavalent syntax to the Matlab version for indicing, however the Julia D4M does have Regex and row/column elementwise indexing.

3. 03math : This showcases mathematical operations that D4M associative array can utilize: element-wise divide, sum on 1 dimension, matrix multiply and others.
4. 04advconstruct : This script tests construction of empty associative array and mixed type associative array.

Parallel Database

Parallel Database is a collection of examples to test D4M with parallel Database. Unfortunately, Julia binding to Accumulo isn't completed yet. Thus this collections of Julia script is implemented upto the point where the database is included.

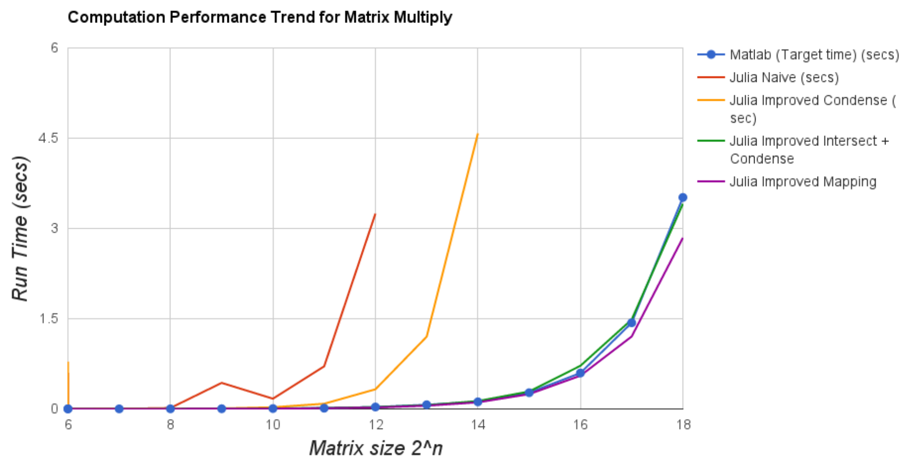
Matrix Performance

Matrix Performance is a collection of examples to test performance of D4M associative array operations. This allows some basic comparsion between Julia and Matlab native operations (like dense and sparse matrix) and D4M associative array operations.

Result

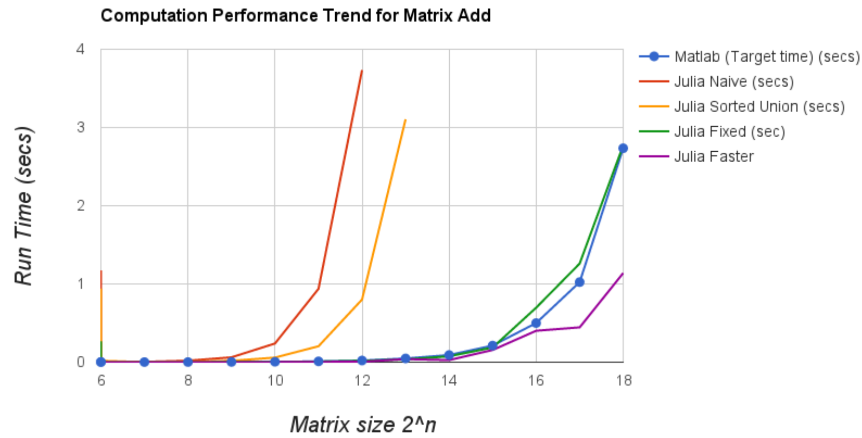
There are two main result of this project. First a base D4M implementation is completed in Github. And that the implementation has performed on par or better than the matlab implementation. The Julia D4M module has been implemented at https://github.com/achen12/D4M.jl.

Matrix Multiply



Intially, the Julia code performed a lot slower than the matlab, but after improving the condensing process and implementing set operations, the Julia implementation achieved performance faster than Matlab. Namely, by assuming the input arrays are unqiue and sorted, the set operation doesn't need to look backward to check previous elements thus resulting in a linear computation speed. These improvements were isolated out as helper functions and aided in speeding up various other operations.

Matrix Add



Matrix add's speed improvement process is similar to matrix multiply. However, during the process a bug in Julia's sparse matrix, in that set index is significantly slower to addition. This could be due to the nature that both right hand side and left hand side are sparse, but a more detailed explanation is needed.

```
%Broken Version
...
ABA = spzeros(n,n)
ABA[Arow,Acol] = At.A
ABA[Brow,Bcol] += Bt.A
...

%Fixed Version
...
ABA = spzeros(n,n)
ABA[Arow,Acol] += At.A %Change
ABA[Brow,Bcol] += Bt.A
...
```

Parallelism?

Why isn't D4M parallelized?

It could be, but not the part that was focused on. In this sense, the majority of the computation cost for matrix add and multiply lies in the sparse matrix computational cost. The cost of doing the mathematical operation on the sparse matrix and condensing the sparse matrix represent most of the computational cost.

The sorted set operations has been is quite optimized. The cost of computing these operation takes less than the cost of relocating the memory between processes. The example parallelization is done as a test.

```
Aref = @spawn searchsortedmapping(ABintersect,At.col)
Bref = @spawn searchsortedmapping(ABintersect,Bt.row)
AintMap = fetch(Aref)
BintMap = fetch(Bref)
```

But the worst case computational cost for all of the sorted operation is the total count of the elements in each input array. Thus the computation cost is proportational to the memory rellocation cost. In fact the computation is trival compare to the memory rellocation of the array. Thus parallelization would only make sense in two scenarios.

First, the parallization lies in Sparse Matrix. Since the majority of the computation is in the sparse matrix and is not linear with the entries, parallizing the sparse matrix would be very benefitial. But of course this is very difficult. There seems to have some success with Parallel Sparse Matrix module in Julia, but it doesn't seem to be stable. And thus this module is not included as a dependency in Julia D4M. Perhaps one day when the module stablizes and is implemented widely as one of the Julia core modules, D4M could opt in to the module.

Second, the memory is parallized. In essence, associative array is implemented as a block matrix implementation. Thus the memeory has be paritioned, and each of the set operations would be done on the correlated core that has the paritioned memory. This type of implementation might be neccesary for the matrix size that is overwhelming large for the native Julia to capture with normal array. But for the demo in Matrix Performance, this is not necessary yet.

Contribution

This project successfully complete these items:

- Implemented base functions of D4M in Julia on par with the performance in Matlab.
- Completed Documentation for these functions.
- Completed Examples for these functions.

Future Work

- Expand D4M functions
- Accumulo binding for Julia
- D4M interaction with Native Julia

There are plenty of potential work for Julia D4M. First, not all of the functions of D4M has been implemented. There are a few more complex functions that can be implemented, which would match the Julia implementation with the matlab in terms of functionality. Second, a lot of work on D4M also utilizes Accumulo. And thus a Accumulo binding for Julia would be extremely useful for past D4M users to migrate to Julia version of D4M, without changing the data storage and memory setup. Third, there could be potential for mix matching operations between Julia native data structures such as dictionary or matrices, to enable intermixing operations. This can be done, because D4M can properly represent these native data structures in D4M associative array and allow potential intermixing with other database interface modules easily through Julia's native data representations.