Distributed Sparse Matrices in Julia

George Xing

December 16, 2011

1 Introduction

Sparse matrices have a broad use in scientific computing, with applications including but not limited to computational fluid dynamics, circuit analysis, and numerical approximations to partial differential equations. They are particularly useful for large-scale problems, where otherwise infeasible operations can be performed by removing the overhead of operating on and storing zeros. For even larger problems, it may be necessary to distribute data over several processors.

In this report, we discuss the design choices behind an ongoing implementation of a distributed sparse matrix type in Julia. We first give a brief overview of the notion of (local) sparse matrices, and Julia's implementation of this. We follow with an exposition of the distributed extension of sparse matrices. We exhibit a possible, if impractical, use case, in the form of solving the minimal cost spanning tree problem. Finally, we conclude with a discussion of possible improvements and future work.

2 Sparse Matrices

2.1 Design Principles

The implementation of sparse matrices in Julia follows principles which adhere closely to those of MATLAB [1]. Given a matrix, there are several possible ways to represent its values. In selecting between these alternatives, several considerations must be made. Memory use should be minimal – the memory used should scale with the number of nonzero elements of the matrix, not its total number of elements. Additionally, the time taken to perform a sparse operation should be proportional to the number of nonzero operations in its equivalent dense operation. In particular, this suggests that we should make it possible to iterate over the nonzero values (and only those) of a sparse matrix. Such a consideration dismisses, for example, a implementation with index tuple keys hashed to their corresponding nonzero values, for to be able to iterate over its keys, we may as well store its keys in an array. Though it is possible to use specialized schemes for differently structured matrices (for example, we can simply store the diagonals of a banded matrix), we make no attempt to do

so for the sake of simplicity. In the end, we used the same storage scheme as MATLAB, the Compressed Sparse Column (CSC) format.

2.2 The Compressed Sparse Column Format

Suppose we had a $m \times n$ matrix A with many zeros, and we'd like to store its nonzero values. Let $nnz(A)$ denote the number of nonzero values of A. A simple method would be to store all indexes where the matrix is nonzero, and their corresponding nonzero values. For example, we could have three arrays `rowval`, `colval`, and `nzval`, each of length $nnz(A)$, where $A[\texttt{rowval}[i], \texttt{colval}[i]] = \texttt{nzval}[i]$, for $1 \leq i \leq nnz(A)$, and every other entry of A is zero. We could also store the indexes in some sorted order; for our purposes, let us order them in dictionary order, first by column index, then by row index. For examples, the 5×4 matrix

$$A = \begin{pmatrix} 1 & 4 & 7 & \\ 2 & & & \\ 3 & & 8 & 10 \\ & 5 & & 11 \\ & 6 & 9 & \end{pmatrix}$$

would be transformed into

$$\begin{aligned} \texttt{rowval} &= [1, 2, 3, 1, 4, 5, 1, 3, 5, 3, 4], \\ \texttt{colval} &= [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4], \\ \texttt{nzval} &= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. \end{aligned}$$

This representation doesn't always save space; notice that in this particular example, the original matrix had size 20, but we used 33 elements to represent it. In general, if we make the conservative assumption that each element in `rowval`, `colval`, `nzval` takes up the same amount of memory (this is often the case in Julia, where 64-bit integers are common, but with 32-bit integer indices and 64-bit floating point numbers, this isn't always the case), then this saves space when less than one-third of the matrix is populated.

We also note that since the `colval` array is simply a rising sequence of the column index, we can save space by replacing it with a `colptr` array of size $n + 1$, such that the indexes from $\texttt{colptr}[i]$ to $\texttt{colptr}[i + 1] - 1$ in `rowval` represent values in the i-th column. In our example above, the corresponding replacement would be $\texttt{colptr} = [1, 4, 7, 9, 12]$.

The three arrays `colptr`, `rowval`, and `nzval` make up the CSC format. (Actually, in our implementation, we have a `nvals` field which stores the number of nonzeros values in our matrix, and only the initial values of `rowval` and `nzval` are used, leaving space for additional storage, if necessary.)

This format has several tradeoffs. On the one hand, iteration over the nonzero values of a column is trivial, which is useful for operations such as matrix-vector or matrix-matrix multiplication. On the other hand, operations like elementwise reference and assign, which would take constant time in a dense representation, take $O(\log nnz)$ and $O(nnz)$ respectively, where nnz denotes the number of nonzero values of the matrix.

3 Distributed Sparse Matrices

For larger problems, it may be desired or even necessary to partition data among several processors. We introduce the `DSparseMat` type, based off of the `DArray` type, which accomplishes this for sparse matrices.

3.1 Distributing a Sparse Matrix

To split up the data of a matrix over several processors, we simply divide it into contiguous block columns and assign a block column to each processor. Each processor also stores several metadata fields, including `pmap` and `dist`, which are arrays of size np and $np+1$, respectively (where np is the number of processors in the distributed sparse matrix), such that the processor with number `pmap`$[i]$ owns columns `dist`$[i]$ through `dist`$[i+1]-1$. These fields allow any processor to be able to find the (other) processors which own the data involved in a query, and send the appropriate messages to recieve this data or have it altered.

3.2 Distributed Operations and Indexing on `DSparseMat` Objects

Our goals in designing a type for a distributed sparse matrix include ease of access and flexibility for the end user. We would like Julia users to be able to think of data accesses in terms of indexing when possible, rather than being locked in a message-passing or other traditional parallel paradigm.

Every time an indexing operation (either `ref` or `assign`) is called on a distributed sparse array, we sort the column indices, and using the `dist` and `pmap` fields, determine which processors have the data we need to construct the matrix. We then send a message (in the form of a `remote_call`, a request to run a function and return its value) to each processor involved. Each processor runs this request locally, which is a local sparse matrix reference or assignment. In the case of `ref`, the results are relayed back to the original processor, which aggregates the data. This aggregation can be done by copying individual columns (in the form of subsections of `rowval` and `nzval` arrays) in the original order of the reference assignment, and combining these to form `rowval` and `nzval` fields of the entire sparse matrix.

Other operations on the entire matrix can have their work distributed across processors. For example, a matrix-vector multiplication can be done by having each processor compute the value of its local piece times the vector, and having all local answers added up in a parallel reduce. More complicated functionality is not yet supported; it is our hope that we have provided a framework which allows others (as well as ourselves) to add this functionality, in both direct and sparse methods.

4 Minimal Cost Spanning Trees

We apply our distributed sparse matrix problem to solving the minimal cost spanning tree problem. As it will turn out, our choice of algorithm, selected for its simplicity and immediate application to this type, will be impractical for even small problem sizes.

4.1 Definitions

We are given an undirected graph $G = (V, E)$ (with vertex set V and edge set E). We call a subset of its edges $C \subset E$ a *cycle* if is equal to the set $\{(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)\}$ for some k and $v_1, v_2, \ldots, v_k$. (We say that $(u, v) = (v, u)$, since the graph is undirected.) We call a subset of its edges $F \subset E$ a *forest* if it contains no cycles. We call a subset of its edges $T \subset E$ a *tree* if it is a forest, and the edges form one connected component (excluding any singleton vertices). Finally, we call a subset of its edges $T \subset E$ a *spanning tree* if it is a tree with cardinality $|V| - 1$; i.e., one that connects all vertices. Note that G is connected if and only if it has a spanning tree.

Given a connected, undirected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}$, we let $c(T) = \sum_{e \in T} c(e)$, for any $T \subset E$. The minimal cost spanning tree problem is to find the minimum value of $c(T)$ over all spanning trees T.

4.2 Prim's Algorithm

The minimal spanning tree problem has been extensively studied and has many algorithms, such as a linear time randomized algorithm due to Karger, Klein, and Tarjan [2], and a deterministic $O(|E|\alpha(|E|))$ time algorithm due to Chazelle [3]. Prim's algorithm, on the other hand, is a simple, $O(|V|^2)$ algorithm. It runs as follows. Given $G = (V, E)$ with cost function c:

- Set $V_{done} = \{v_0\}$ for an arbitrary $v_0 \in V$. Set d to be an array indexed by the vertices, with all values initialized to ∞. Set $cost = 0$.
- For $v \in V\{v_0\}$ such that $(v, v_0) \in E$, set $d[v] = c(v_0, v)$.
- While $V_{done} \neq V$:
  - Set $u = \text{argmin}\{d[v] | v \in V \backslash V_{done}\}$.
  - Set $V_{done} = V_{done} \cup \{u\}$.
  - For $v \in V \backslash V_{done}$ such that $(v, u) \in E$, set $d[v] = \min(d[v], c(u, v))$.
  - Set $cost = cost + d[u]$.

At a high level, this algorithm maintains V_{done}, a set of vertices initialized to be an arbitrary vertex, and at each step finds the vertex not in V_{done} closest to it, and adds this vertex (incrementing the total cost of the tree appropriately), doing this until there are no vertices to be added.

4.3 Implementing Prim's Algorithm with Distributed Memory

To start, we may assume that all costs in the graph are positive by adding a sufficiently large cost M to each edge of the graph. This changes the result only by adding $(|V| - 1)M$ to the cost, which we can simply subtract at the end. Then, we may represent our graph $G = (V, E)$ as an adjacency matrix A, where $A[i, j] = A[j, i] = c(i, j)$ if $(i, j) \in E$, and $A[i, j] = A[j, i] = 0$ otherwise. By representing A as a sparse matrix, we are able to easily iterate over the neighbors of any vertex: they are simply represented by the nonzero values in a column.

Our parallel implementation simply does the following: it distributes A and d across all processors. To find u, the next closest-vertex, is an argmin operation over the distributed array d; this is done with a parallel reduce, and has an asymptotic runtime of $O(\log p)$, where p is the number of processors. Once this minimal vertex is found, the result is sent to all processors, which do the corresponding updates locally. In total, the serial $O(|V|^2)$ work is split among the p processors, and there are $|V|$ steps of communication, giving a total runtime of $O(|V|^2/p + |V| \log p)$.

4.4 Results and Discussion

Below, we show the results of our parallel implementation on 1 (a serial version), 2, and 4 processors on varying problem sizes (shown are the values of $|V|$, with $|E| \approx \frac{1.1 \ln |V|}{|V|}$).

	1000	2000	3000	4000	5000
1	0.00484	0.018801	0.054413	0.0743	0.11729
2	4.5325	9.608	14.9646	27.6719	33.3689
4	51.071	95.141	141.206	193.537	234.467

It's clear that this parallel algorithm performs quite abysmally compared to a serial version, going about 284 times slower for 2 processors than in serial, and even experiencing a parallel slowdown when adding more processors. Our choice of algorithm was poor in this case; for each round, the time taken to do the local search and update is completely dominated by the communication overhead.

For the specific case of this problem, it would likely be better to use a form of Borůvka's algorithm, which maintains a forest of connected components, and can independently add edges to each. This would allow for more local computation before communication, cutting down on the overhead.

5 Future Work

I plan to continue to work on the Julia project, maintaining and adding functionality to the sparse matrix and distributed sparse matrix classes. Julia's current functionality on these fronts is, for lack of a more fitting word, sparse.

I hope to soon integrate SuiteSparse [4], Tim Davis's large C library for direct sparse methods. From here, I will possibly round out more functionality in the sparse classes, as well as optimize existing code for competitiveness with MATLAB. An eventual goal of mine would be to implement some distributed direct methods, such as a parallel LU or Cholesky decomposition; this, however, would be a quite nontrivial task.

6 Acknowledgements

Many thanks to Prof. Alan Edelman, whose interesting, insightful, and humorous lectures helped get me interested in sparse matrices. Also many thanks to the original developers of Julia: Jeff Bezanson, Viral Shah, and Stefan Karpinski. Special thanks go out to Jeff, whose `DArray` class I borrowed from, and Viral, who was responsible for many of the inital sparse matrix features.

References

[1] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 1, 1992

[2] Karger, David R.; Klein, Philip N.; Tarjan, Robert E., "A randomized linear-time algorithm to find minimum spanning tree", *Journal of the Association for Computing Machinery*, Vol. 42 No. 2, 1995

[3] Chazelle, Bernard, "A minimum spanning tree algorithm with inverse-Ackermann type complexity", *Journal of the Association for Computing Machinery*, Vol. 47, No. 6, 2000

[4] Davis, Tim, "SuiteSparse", `http://www.cise.ufl.edu/research/sparse/SuiteSparse`.