Summary: The Combinatorial BLAS: Design, implementation, and applications. (2011) by Aydin Buluc (Lawrence Berkeley National Laboratory) and John R. Gilbert (University of Santa Barbara, my advisor! :) )

Main point: The Combinatorial BLAS is a scalable, high-performance software library for graph analysis. Linear algebraic primitives, such as sparse matrix-vector multiplication, are shown to be the key for parallelization of graph computations.

The Combinatorial BLAS (CombBLAS) builds upon the ideas of the Basic Linear Algebra Subroutines (BLAS), which are low-level subroutines common to many linear algebra computations (for example, a low-level subroutine is matrix-vector multiplication, which is common to many numerical linear algebra algorithms such as the Lanczos algorithm for finding eigenvalues and eigenvectors of a matrix). CombBLAS is thus a composed of a set of primitives that are common to many graph computations. For example, traversing all the the edges incident to a vertex is a primitive used by breadth-first search. CombBlas is sparse-array-based, meaning that graph computations are performed on sparse matrices representing the connectivity of the graph. This approach is in contrast to the visitor-based primitives used by, for example, the Boost Graph Library, which “describe operations one vertex or edge at a time.” While powerful for sequential algorithms, visitor-based approaches are difficult to parallelize.

The most important piece of CombBLAS are distributed sparse matrices, which represent a graph’s structure. The three other objects present in the library are dense matrices, dense vectors, and sparse vectors. The subroutines are: sparse matrix-sparse matrix multiplication, sparse matrix - (sparse or dense) vector multiplication, sparse element-wise multiplication, reduction of dimension (for example, sum the columns into a row vector), sparse matrix reference (refer to a sub-matrix of a sparse matrix), sparse sub-matrix assign (assign a sub-matrix of a sparse matrix to new values), scale (scale the matrix by rows, columns, or element-wise), sparse element-wise application (apply a unary operator to all nonzeros in a sparse matrix). These routines were chosen by following these guiding principles:

• If multiple operations can be handled by a single function prototype without degrading the asymptotic performance of the algorithm they are to be part of, then provide a generalized single prototype. Otherwise, provide multiple prototypes

• If an operation can be efficiently implemented by composing a few simpler operations, then do not provide a function for that operator

• For operations that can be implemented in place, deny access to any other variants only if those increase the running time (ex: element-wise apply)

• In-place operations have slightly different semantics depending on whether the operands are sparse or dense. Semantics favor leaving the sparsity
pattern of the underlying object intact as long as another function handles the more conventional semantics that introduces/deletes nonzeros

The data structure used to implement distributed sparse matrices is the *SpDistMat* object with base class *SpMat*. Types of objects are the same as their corresponding base classes, and static object oriented programming techniques are used in favor of dynamic dispatch (in other words, the correct implementation of a polymorphic operation is determined at compile time instead of at runtime). The implementation of the *SpMat* base class includes an “essential” parameter that specifies the size, number of nonzeros, and starting addresses of the internal arrays, along with their sizes, that describe the matrix. Since sparse matrix storage formats are composed of arrays, this allows for efficient communication and computation on those arrays. The functions inside the *SpMat* include can create a new sparse matrix, multiply it by another sparse matrix, insert elements into the matrix, split or merge the matrix, transpose the matrix, get its essentials or arrays, write and read a matrix to/from a file, and give a particular row, column, or all the non-zeros.

In the distributed setting, the matrix is distributed across processors in a 2-D block composition. In the dense case, this decomposition corresponds to a $\sqrt{p} \times \sqrt{p}$ logical grid of processors, where each processor gets a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ piece of an $n \times n$ matrix. In the sparse case, CombBLAS uses a doubly-compressed sparse column (DCSC) format to store the sparse matrix. This format uses $O(nnz)$ storage, and is similar to the compressed sparse column format, as described in Buluc and Gilbert’s *Highly Parallel Sparse Matrix-Matrix Multiplication*.

The performance gains of CombBLAS is described by showing performance results for two applications: breadth-first search and Markov clustering. Both show dramatic speedup and scalability.

Markov clustering, in broad terms, seeks to identify clusters by simulating a random walk on a graph and identifying regions where the walk is likely to stay. This is done by iterating two steps iteratively. At iteration $i$, 1) the transition probability matrix $P$ on the graph is raised to the $i$th power, and 2) the entries inside $P$ are raised to a power and set to 0 if they fall below a certain threshold. This process is clearly amenable to a linear-algebraic graph library implementation. With a CombBLAS implementation, Markov clustering is achieved on RMAT graphs of up to 4096 processors with very good strong scaling results.

The authors conclude by pointing out the utility of CombBLAS in being the middleware between discrete structures (graphs) and their computer implementations. They point out that the main contributions of CombBLAS are the guiding principles and primitives, rather than the explicit implementation of the library.