A Hybrid GraphBLAS in C++11: specification, design, implementation, and performance

A. N. Yzelman (Huawei Zürich Research Center) and W. J. Suijlen (Huawei Paris Research Center)

{albertjan.yzelman,wijnand.suijlen}@huawei.com

Objectives

Provide a solution for graph computations which

- has fast shared- and distributed-memory performance, without requiring expert HPC programming;
- resolves as much as possible at compile-time, reducing run-time overheads and rejecting malformed programs;
- runs not only stand-alone, but is able to integrate with existing HPC or Big Data frameworks;
- deploys on traditional x86 HPC clusters, but also on edge devices (Android on ARM64);
- allows transparent offloading to accelerators, not requiring any changes to GraphBLAS algos; and
- allows design of (graph) algorithms guaranteed to adhere to strict asymptotic performance bounds, never incurring unexpected performance.

This is joint work with J. M. Nash and Daniel Di Nardo.

Introduction

The GraphBLAS provides a rigorous mathematical basis for graph computations [1], allowing for a simple **data-centric** expression of graph and sparse matrix computations. It conveniently allows hiding a significant amount of know-how for high-performance sparse matrix computations [2].

GraphBLAS exploits the relationship between sparse linear algebra over **generalised semirings** and graph algorithms. In our formalisation, such a semiring consists of four domains $D_1, D_2 \ldots, D_4$, two binary operators $\oplus : D_3 \times D_4 \to D_4$ and $\otimes : D_1 \times D_2 \to D_3$, and two identities $\mathbf{0}$ and $\mathbf{1}$. The choice of operators and identities under the chosen domains should adhere to the common rules regarding associativity, commutativity, identity, distributivity, and annihilation. E.g.:

- $(\mathbb{Z}, +, \cdot, 0, 1)$: natural numbers and regular add/multiply;
- $(\{false, true\}, \vee, \wedge, false, true)$: the boolean semiring;
- $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$: the min tropical semiring.

The additive operator $\oplus$ with its $\mathbf{0}$-identity forms a commutative monoid, while the multiplicative operators $\otimes$ with $\mathbf{1}$ forms a regular monoid.

Kepner and Gilbert describe how **sparse linear algebra** over various semirings can express various graph algorithms [3]. Graphs are respresented by sparse matrices where edges correspond to nonzero coordinates and edge weights correspond to nonzero values. Sets of vertices (and their weights) correspond to nonzero indices (and their values) in sparse vectors. Using mathematical concepts explicitly in programming also applies to general (parallel) programming [4].

Specification

We expose two containers:

- grb::Vector< T >, grb::Matrix< T >

Data ingestion and extraction occurs via STL-compatible **iterators**. Operators $f : D_1 \times D_2 \to D_3$ are templated classes:

- grb::operators::add< $D1, D2, D3$ > f; $(f(x, y) = x + y)$.

Identities are similarly templated:

- grb::identities::zero< T >, grb::identities::infinity< T >

Identities and operators can be **composed**:

- grb::Monoid< $D1, D2, D3$,
  grb::operators::add, grb::identities::zero>
- grb::Semiring< $D1, D2, D3, D4$,
  grb::operators::add, grb::identities::zero,
  grb::operators::multiply, grb::identities::one>

We expose GraphBLAS **type traits** such as:

- grb::is_object< T >, grb::is_container< T >
- grb::is_operator< T >, grb::is_monoid< T >
- grb::is_commutative< OP >, grb::is_associative< OP >

We allow **easy extension** beyond the 13 standard operators, and allow stateful operators, monoids, and semirings.

GraphBLAS primitives operate on scalar values and/or containers under a given operator, monoid, or semiring. Some examples and their effect, organised by their BLAS-like level:

- level 0: scalar, grb::foldl(y, x, f), $y \leftarrow f(x, y)$;
- level 1: vector–vector, grb::foldl(α, x, monoid), $\sum_i x_i$;
- level 2: matrix–vector, grb::mxv(y, A, x, ring), $y = Ax$;
- level 3: matrix–matrix, grb::mxm(C, A, B, ring), $C = AB$;

Output scalars or containers are the first argument to a primitive. Output containers may be **masked**, and the output of an operation may be **accumulated** into any existing content. Masks and accumulators may be omitted when unused; i.e., the following two mxv ($y = Ax$) calls are equivalent:

- const auto &accum = ring.getAdditiveOperator();
- grb::mxv(y, NO_MASK, accum, A, x, ring);
- grb::mxv(y, A, x, ring);

Descriptors reinterpret arguments to primitives. E.g.,

- grb::mxv<grb::descriptors::transpose |
  grb::descriptors::no_casting> (y, A, x, ring);

computes $y = A^T x$ and disallows semirings whose domains mismatch vector element or matrix element types.

Each primitive must have **explicit performance semantics** in terms of work, local data movement, network data movement, and storage. Primitives may not make system calls.

Example: k nearest neighbours

This algorithm is called on an input matrix A, a *source* vertex, a number of hops k, an output vector u, and a user-defined descriptor *descr*. It demonstrates descriptors and **masks**:

- grb::Semiring<bool, bool, bool, bool,
  grb::operators::logical_or, grb::operators::logical_and,
  grb::identities::logical_false,grb::identities::logical_true>
  ring;
- grb::Vector<bool> x(grb::ncols(A));
- grb::RC err = grb::setElement(x, *true*, *source*);
- if err does not equal grb::SUCCESS, return err;
- return grb::mpv< descr | grb::descriptors::add_identity
  > (u, A, k, x,ring); //computes $u = (A + I)^k x$

The last line may be replaced with k iterations over

- grb::mxv< descr | grb::descriptors::invert_mask>
  (u, x,ring.getAdditiveOperator(), A, x, ring);

Results

We ran the canonical PageRank algorithm on various graphs using a Big Data framework interfacing with a distributed storage system, comprising a common contemporary use of parallel computing. Let a Spark user execute:

- val output = GraphBLAS.pagerank(context,filename);

In *Pure Spark*, the above function calls a Spark PageRank implementation with data loaded from HDFS. *Accelerated Spark* offloads the computation to our hybrid GraphBLAS implementation and employs parallel I/O instead, both when loading and when returning the PR vector:

		Pure (s.)		Accelerated (s.)	
	GB	load	full PR	load	full PR
cage15	2.5	39.6	1296.9	5.5	19.2
uk-2002	4.7	168.6	>4 hrs	8.7	48.7
clueweb12	786	-	-	658.8	1875.0

Design

Every container in our implementation requires two template arguments, the second being the **backend**. Primitives are overloaded for different backends, thus providing different implementations for different target architectures. Compilation occurs with a default backend selected.

We implement three backends: **sequential**, **threaded**, and **distributed**. In the latter case, we define two I/O modes: sequential (iterators touch all available data) vs. parallel (iterators touch local data only).

Implementation

The sequential backend uses **Gustavson's data structure** (CRS+CCS) to store graphs. The nonzero structure of a (sparse) vector are maintained via a boolean array and a stack of indices. Dense level-1 primitives map to **vectorised instructions** using all tested compilers and architectures.

The threaded backend employs **OpenMP** to parallelise the sequential one. Concurrent stack updates occur by updating local stacks, followed by a prefix-sum on local stack sizes used to copy local entries into a global stack. Microbenchmarks guide the choice of scheduling policies for different operations.

Data ingestion into the distributed backend uses a block-cyclic distribution for load-balancing. Internally, we employ a **1D row-wise block distribution**. Vectors may require synchronisation on input or output with SpM(Sp)Vs– if dense, this requires allreduces/allgathers on value arrays; while for sparse vectors we either synchronise using the boolean arrays or using the index stacks, whichever incurs least communication. We use **Lightweight Parallel Foundations** [5] for communication with asymptotic performance guarantees, and use either the sequential or threaded backend for local computations– the latter choice resulting in a **hybrid GraphBLAS**.

Future Directions

While a 1D distribution suffices for relatively small scales, we like to use 2D partitioning using Mondriaan, PaToH, Zoltan, etc.; but **API changes may be necessary**. Other ideas:

- performance modeling for better choice of synchronisation;
- incorporate more of past lessons learned [2, 6];
- a storage-optimal back-end (we optimised for speed); and
- our level-3 primitives require further attention.

References

[1] J Kepner, D Bader, A Buluç, J Gilbert, T Mattson, and H Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science*, 51:2453–2462, 2015.

[2] A N Yzelman and D Roose. High-level strategies for parallel shared-memory sparse matrix–vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, 2013.

[3] J Kepner and J Gilbert. *Graph algorithms in the language of linear algebra.* SIAM, 2011.

[4] A A Stepanov and P McJones. *Elements of programming.* Addison-Wesley Professional, 2009.

[5] W Suijlen and A N Yzelman. Lightweight parallel foundations: a model-compliant communication layer. *arXiv preprint arXiv:1906.03196*, 2019.

[6] A N Yzelman. Generalised vectorisation for sparse matrix–vector multiplication. In *Proceedings of the 5th Workshop on IA^3*, pages 1–8, 2015.