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,wijand.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;
develops on traditional s86 HPC clusters, but also on edge devices (Android on ARM64);
allows transparent offloading to accelerators, not requiring any changes to GraphBLAS algo’s;
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 \( D_1, D_2, \ldots, D_n \), two binary operators \( \otimes \) and \( \oplus \), and two identities 0 and 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.,

\[
(\otimes, +, \cdot, 0, 1) \quad \text{natural numbers and regular add/multiply;}

(\{false, true\}, \land, \lor, false, true) \quad \text{the boolean semiring;}

(\{min, +\}, min, +\infty, 0) \quad \text{the minimal tropical semiring.}
\]

The additive operator \( \oplus \) with the \( 0 \)-identity forms a commutative monoid, while the multiplicative operators \( \otimes \) with 1 forms a regular monoid.

Kepner and Gilbert describe how sparse linear algebra over various semirings can express various graph algorithms [5].

Graphs are represented 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:

- \( \text{grb::vector<V, T>} \)
- \( \text{grb::matrix<T>} \)

Data ingestion and extraction occurs via STL-compatible iterators. Operators \( f_1 \times D_1 \rightarrow D_2 \rightarrow D_3 \) are templated classes:

- \( \text{grb::operators::add} \rightarrow D_1, D_2, D_3 \rightarrow f, (f(x, y)) = x + y \).
- \( \text{grb::identities::zero} \rightarrow D_1 \rightarrow f, (f(x, y)) = x \).
- \( \text{grb::operators::multiply} \rightarrow D_1 \rightarrow f, (f(x, y)) = x \cdot y \).

Identities are similarly templated:

- \( \text{grb::identities::zero} \rightarrow D_1 \rightarrow f, (f(x, y)) = x \).
- \( \text{grb::identities::one} \rightarrow D_1 \rightarrow f, (f(x, y)) = y \).

We expose GraphBLAS type traits such as:

- \( \text{grb::is_object<T>} \rightarrow T \)
- \( \text{grb::is_container<T>} \rightarrow T \)
- \( \text{grb::is_commutative<T>} \rightarrow T \)
- \( \text{grb::is_monoid<T>} \rightarrow T \)
- \( \text{grb::is_semiring<T>} \rightarrow T \)

We allow easy extension beyond the standard operators, and allow standard 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:

- \( \text{grb::mxv} \)
- \( \text{grb::tuxv} \)
- \( \text{grb::euxv} \)
- \( \text{grb::huxv} \)

Identities and operators can be composed:

- \( \text{grb::operators::add} \)
- \( \text{grb::operators::multiply} \)

The additive operator \( \oplus \) with its \( 0 \)-identity forms a commutative monoid, while the multiplicative operators \( \otimes \) with 1 forms a regular monoid.

Kepner and Gilbert describe how sparse linear algebra over various semirings can express various graph algorithms [5].

Graphs are represented 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].

Example: \( k \) nearest neighbors

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 \( \text{desc} \).

It demonstrates descriptors and masks:

- \( \text{grb::semiring<bool, bool, bool, bool>} \)
- \( \text{grb::operators::logical_or, grb::operators::logical_and} \)
- \( \text{grb::identities::false, grb::identities::logical_true} \)
- \( \text{ring} \)

- \( \text{grb::vector<T, bool>} \rightarrow \text{grb::bool}(A) \)

- \( \text{grb::rc} \rightarrow \text{getLevel}(\text{r, true, source}) \)

- \( \text{grb::rc} \rightarrow \text{getLevel}(\text{r, true, source}) \)

- \( \text{grb::rc} \rightarrow \text{getLevel}(\text{r, true, source}) \)

The last line may be replaced with \( k \) iterations over:

- \( \text{grb::mxv} \rightarrow \text{grb::descriptors:invert_mask} \)
- \( (x, x, \text{getAdditiveOperator}, A, x, x) \).

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. Computation 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 Gustavsson’s data structure (CSR+CCS) to store graphs. The nonzero structure of a (sparse) vector are maintained via a boolean vector 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 task 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 all-reduces/allgathers on value arrays, while for sparse vectors we either synchronise using the boolean arrays or using the index stacks, whichever incurs less 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 Mondrian, 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


