A C++ GRAPHBLAS: SPECIFICATION, IMPLEMENTATION, PARALLELISATION, AND EVALUATION

A. N. YZELMAN, D. DI NARDO, J. M. NASH, AND W. J. SUIJLEN

Abstract. The GraphBLAS is a programming model that expresses graph algorithms in linear algebraic terms. It takes an easy-to-use, data-centric view where algebraic operations execute over sparse or dense vectors and sparse matrices, parametrised in the algebra the computation should proceed with. We present a C++ GraphBLAS interface designed with both shared- and distributed-memory systems in mind. Our specification employs STL-compatible generic programming principles. Templates and type traits allow for enhanced compile-time error handling and optimisation, even for user-defined types and operators. Our specification guides algorithm designers towards writing performant code by attaching performance semantics to each operation.

We detail a reference implementation that auto-vectorises and auto-parallelises user programs for sequential as well as shared- and distributed-memory parallel machines. We demonstrate the use of our API via two canonical graph algorithms, the k-nearest neighbourhood and PageRank. Weak and strong scaling experiments using real-world datasets with up to 42 billion edges show that the defined performance semantics are achievable in practice and capture key performance characteristics of the two selected algorithms. Microbenchmarks and throughput estimates from the PageRank algorithm on real-world graphs furthermore show that our reference implementation can extract near peak performance on different architectures.

1. Introduction

The GraphBLAS is a recent standard [KBB+15, BMM+17] for expressing graph computations in the language of linear algebra, a concept first explored by Kepner and Gilbert [KG11], Gilbert and Buluç [BG11], and Lugowski et al. [LBGR12]. The GraphBLAS distinguishes three core concepts: algebraic containers, algebraic relations, and algebraic operators. Sparse matrices and vectors are an example of the former, while the sparse matrix–vector (SpMV) multiplication is an example of the latter. A generalised semiring under which an SpMV multiplication takes place is one example of an algebraic relation. The operations the GraphBLAS defines join the three concepts by operating on containers, while taking binary operators, monoids, or semirings as arguments. Exposing these three concepts explicitly allows us to represent canonical graph algorithms in linear algebraic terms. By using a min-plus algebra, for example, repeated SpMV multiplication becomes equivalent to the Δ-stepping algorithm for solving the single-short single paths problem [MS03], while a single sparse matrix–sparse matrix multiplication using the same algebra would solve the all-pairs shortest paths problem.

This work presents our C++11 alternative of the C GraphBLAS standard [BMM+17], which takes the core concepts of GraphBLAS and unifies them with standard C++ principles. Our systematic use of templates and type traits allow additional error handling and optimisations to occur at compile time instead of at run time, while we define I/O involving GraphBLAS containers using standard template library (STL) compatible iterators. User-defined operators and user-defined types follow naturally from the use of templates. We furthermore expose more properties of algebraic relations, allowing GraphBLAS implementation additional transformations that benefit performance. These primarily takes place through *type traits* which allow compile-time introspection of properties of the algebraic relations used within the GraphBLAS. Type traits also improve user feedback: a GraphBLAS implementation can inspect whether a given operator is associative when required, and if not, emit a compile-time error. Finally, we attach cost semantics

Key words and phrases. Graph algorithms, linear algebra, C++, data-centric, auto-parallelisation, auto-vectorisation, shared-memory, distributed-memory, cost modelling.

to each primitive in terms of work, data movement, and number of element-wise operator applications. This allows users to associate costs to GraphBLAS algorithms, thus encouraging them to minimise the resulting upper bound on performance. The specified bounds are asymptotic and worst-case: implementations are encouraged to incur less costs in practice through optimisations and transformations, but are not allowed to perform asymptotically worse than specified.

Our vision consists of a single and simple to use domain-specific library (DSL). This should hide implementation details pertaining to vectorisation and parallelisation necessary for achieving high performance and hide the complexity of dealing with different architectures, yet expose just enough performance characteristics to allow users to gauge the efficacy of the different optimisation trade-offs different implementations may choose. We therefore include and describe a compile-time backend selection mechanism that transparently allows the exact same GraphBLAS code to defer to different implementations. This has two main uses: 1) a specific hardware accelerator can transparently be used via a specially designed backend, and 2) different use cases may employ a specific set of operations or a specific type of input graph, and would hence benefit from differently tailored implementations.

We present three backends: one that vectorises to optimally use SIMD units available on most modern processors, another to exploit modern multi-core processors via OpenMP, and a third for distributed-memory systems via the Lightweight Parallel Foundations (LPF) [SY19]. To support distributed-memory use of GraphBLAS algorithms we introduce the notion of user processes, extensions for parallel I/O, and define a decentralised error handling mechanism. Backends furthermore are composable: our distributed-memory backend parallelises operations by decomposing them into process-local ones interspersed with inter-process communication, and can use the shared-memory backend to enable process-local computations. Thus by mixing backends, we obtain a fully hybrid shared- and distributed-memory GraphBLAS backend.

We describe our C++ GraphBLAS API in Section 2 while Section 3 details the implementation of our three backends. Section 4 evaluates our specification and our implementations by implementing two canonical graph algorithms and by performing scalability and throughput oriented experiments of these algorithms, using each of our backends. These show that our API allows simple expressions of standard graph algorithms, that costs can easily and systematically be obtained, that execution complies with the derived costings, that our implementation attains near-peak performance on two x86 architectures, and that our hybrid shared- and distributed-memory parallel backend successfully scales up to 42-billion edge graphs on up to six nodes. We relate our specification and implementation to related work in Section 5, while Section 6 concludes. Section 7 presents future work and challenges.

2. A C++ GraphBLAS Interface

This section describes our GraphBLAS API for use with C++. We adhere to the ANSI C++11 standard since the API presented here does not rely on any new features or extensions introduced in later versions. In turn, we discuss:

- user processes, initialisation and finalisation;
- containers, vectors and matrices;
- IO, ingesting data into containers and extracting data from containers;
- expressing algebraic relations, from binary operators to semirings;
- primitives, operations on containers and scalars;
- modifying output, masks and accumulators; and
- descriptors, reinterpreting arguments and slightly modifying behaviour.

All GraphBLAS types and primitives reside in the *grb* namespace. Only the constructors of GraphBLAS containers and certain IO primitives[1] may throw C++ exceptions; all remaining primitives are exception-free. This enables usage of our API when run-time support for C++ exceptions is unavailable. Primitives that cannot fail do not return error codes, while others may return one of from Table 1. All such primitives can return *SUCCESS*. The *ILLEGAL* error code

[1] i.e., those IO primitives that take user-supplied iterators as arguments

Error code	Description
SUCCESS	The primitive completed successfully
ILLEGAL	An illegal argument or illegal combination of arguments
MISMATCH	Mismatching vector and/or matrix dimensions
UNSUPPORTED	The selected backend does not support a request
OUT_OF_MEMORY	Not enough memory available to complete a request
IO_ERROR	Indicates bad input or a failure of an input iterator
PANIC	An unmitigable error has occurred

TABLE 1. List of error codes that exist in our C++ GraphBLAS API

indicates an error in the use of the primitive, while *MISMATCH* indicates the specific error of using vectors and/or matrices in a single algebraic operation while their dimensions do not match. *UNSUPPORTED* indicates a deployment error, for example resulting from making distributed-memory requests from a shared-memory backend. All aforementioned errors reappear should a user attempt the exact same call again. The *OUT_OF_MEMORY* and *IO_ERROR* codes, in contrast, depend on run-time conditions; these errors may be mitigated by freeing up memory or selecting a different input file, for example, and then retrying the call.

The *PANIC* error code signals the GraphBLAS implementation has entered an unrecoverable state: after having received this error code, any future calls to the GraphBLAS API other than to grb::finalize and destructors of GraphBLAS containers result in undefined behaviour. Not aborting immediately allows users to clean up and exit gracefully. Should a primitive return anything other than *SUCCESS* or *PANIC*, the call will never produce any other side effects beyond returning the given error code.

While describing the various GraphBLAS types and primitives, we specify either big-Theta (Θ) or big-Oh ($\mathcal{O}$) bounds on each of the computation times required, the amount of data movement incurred, and the amount of storage a call requires. An implementation shall never exceed these bounds. We also specify which functions may allocate dynamic memory, which may free memory, and which shall never make any such system calls; in performance-critical code, only functionality in the latter category should be employed.

2.1. **User processes, initialisation, and finalisation.** A GraphBLAS context is initialised via a call to grb::initialize and closed via a call to grb::finalize. The former primitive allocates any necessary process-local buffers while the latter releases such resources. The grb::initialize has two optional parameters, integers s and P, which have 0 and 1 as their respective default values. Following a Single Program, Multiple Data (SPMD) paradigm, P here is the number of *user processes* executing the same GraphBLAS program while $s \in \{0, 1, \ldots, P-1\}$ is the ID of the process calling the initialisation routine. Exactly P such calls must be made, each with unique values for s: i.e., grb::initialize must be called *collectively*. Initialisation shall allocate $\mathcal{O}(P+T)$ memory per process, where T is the maximum number of threads supported by this process. The finalize primitive takes no arguments and frees all process-local buffers. Both primitives shall complete in $\mathcal{O}(P+T)$ time.

Initialisation may fail for a number of reasons. When $s \geq P$ the *ILLEGAL* error code is returned. If the backend does not support a given number of processes, for example when requesting $P > 1$ while using a sequential implementation, *UNSUPPORTED* is returned. If the system does not have enough memory for local buffers available, *OUT_OF_MEMORY* is returned. If initialisation should fail for unmitigable reasons, such as for example a failure to initialise the system's ibverbs driver, *PANIC* is returned instead.

A GraphBLAS implementation hence never manages processes: the initialisation and finalisation constructs all assume pre-existing sequential or parallel contexts which a user then associates to a GraphBLAS context– this is why we denote these by *user* processes. For $P > 1$ and during the execution of a GraphBLAS program, error codes are always local and should be mitigated locally. One process may encounter a *PANIC* while others do not: this error propagates only

when inter-process communication is necessary but cannot complete. Thus in any non-trivial application, if one process incurs a *PANIC* code, that error state will eventually propagate to all user processes. This 'lazy' error signalling mechanism ensures a multi-process implementation need not synchronise after each primitive.

2.2. **Containers.** We define two opaque containers: vectors (grb::Vector< T >) and matrices (grb::Matrix< T >). Vectors can be sparse or dense, while matrices are assumed sparse always. Vector and matrix entries are of a static type T, which must be any plain-old-data (POD) type– meaning values must be copyable by byte-for-byte transfers, never requiring any form of marshalling or serialisation. We allow *void* T to indicate pattern vectors and pattern matrices.

The sizes of vectors or matrices containers are fixed on initialisation. A grb::Vector constructor takes its size as a single argument, while a grb::Matrix constructor takes two arguments: its number of rows and columns. Memory allocations for vectors only ever occur during construction, including allocations for process- and thread-local buffers necessary to support any primitive that may take the vector-under-construction as an argument. Allocations for matrix storage likewise may happen during construction of a grb::Matrix, including any required buffer allocations. Further allocations may occur on a call to grb::resize(A, nz), however, where A is a grb::Matrix and nz the maximum number of nonzeroes A may store.

The two container constructors, the grb::resize, and the grb::initialize are the only places where an implementation is allowed to allocate memory. Memory may likewise only be freed when containers are destroyed, during a call to grb::resize, or to grb::finalize. Vector and matrix construction proceeds in $\Theta(n+P)$ time and may require up to $\mathcal{O}(n+P)$ storage and data movement, where n is the vector size or the sum of the number of rows and columns of the matrix. The grb::resize incurs $\Theta(nz)$ time, storage, and data movement. The run-times of both constructors, grb::resize, container destructors, grb::initialize, and grb::finalize are furthermore subject to the time required to make system calls; as such, these calls should be used as sparingly as possible. All other GraphBLAS primitives are guaranteed to never make system calls and hence suitable for use in performance-critical code sections.

The size of a vector is queried by grb::size while that of a matrix may be obtained via grb::nrows and grb::ncols. The grb::nnz returns the number of nonzeroes for both vector and matrix arguments. Each of these four primitives take a vector or a matrix as their only argument, and return the requested value as a *size_t*. The call shall complete within $\Theta(1)$ time and data movement, even if $P > 1$.

2.2.1. *Backends.* An implementation targeting multi-core CPUs will not have much in common with one targeting GPUs– yet both may be required at run-time. To facilitate having multiple such implementations available, GraphBLAS containers have one hidden template parameter beyond the nonzero type T: the *backend identifier*, which defaults to a pre-configured value during compilation. GraphBLAS primitives that take a GraphBLAS container as argument hence must be implemented separately for every backend, thus controlling which implementation is referred to. A valid implementation shall provide at least one backend. Users are never required to provide a backend template argument explicitly.

2.3. **IO.** Ingesting data into GraphBLAS containers occurs via C++ forward iterators. Standard C pointers are valid forward iterators. For sparse data, separate iterators are expected for each index and each value. Vector indices or matrix row indices thus must be given via the i and i_end iterators, matrix column indices via the j and j_end iterators, and values via an v, v_end iterator pair. The two or three containers the iterators refer to must contain an equal number of elements or *IO_ERROR* will be returned.

Input either occurs in *SEQUENTIAL* or *PARALLEL* mode. In the former mode, each user process's iterator is assumed to contain the exact same elements. In the latter, each user process is assumed to have disjoint input instead. The signatures of the input functions are as follows:

- grb::buildMatrix(A, i, i_end, j, j_end, v, v_end, mode);
- grb::buildVector(x, i, i_end, v, v_end, mode);

grb::operators::add< T1, T2, T3 >	$f(x,y) = x + y$
grb::operators::mul< T1, T2, T3 >	$f(x,y) = x \cdot y$
grb::operators::min< T1, T2, T3 >	$f(x,y) = min\{x,y\}$
grb::operators::max< T1, T2, T3 >	$f(x,y) = max\{x,y\}$
grb::operators::sub< T1, T2, T3 >	$f(x,y) = x - y$
grb::operators::div< T1, T2, T3 >	$f(x,y) = x/y$
grb::operators::is_equal< T1, T2, B >	$f(x,y) = \begin{cases} true \text{ if } x{=}y \\ false, \text{ otherwise.} \end{cases}$
grb::operators::logical_and< B, B, B >	$f(x,y) = x \wedge y$
grb::operators::logical_or< B, B, B >	$f(x,y) = x \vee y$

TABLE 2. Standard operators available in GraphBLAS (left) and its behaviour in terms of a function f (right). Here, $x \in T1$, $y \in T2$, $f(x,y) \in T3$, and $B = \{false, true\}$.

These primitives do not allow duplicate entries. For vector or matrix with *void* element types the v and v_end iterators need not be given, while for non-*void* containers the elements of v will be cast to the vector's element type during input. Before a call to grb::buildMatrix, the user must have called grb::resize with nz equal or larger than the number of elements the iterators span. Both primitives complete in $\Theta(nz)$ time and data movement, though the time required for dereferencing, incrementing, and copying the supplied forward iterators cannot be guaranteed by GraphBLAS.

Both grb::Vector and grb::Matrix expose the begin(), cbegin(), end(), and cend() public member functions. All four functions return const-iterators to the container's elements and take the IO mode as an optional argument. In *SEQUENTIAL* mode, each user process obtains an iterator over all elements in the container, while in *PARALLEL* mode each user process obtains an iterator over a disjoint part instead. Obtaining an iterator in parallel mode requires $\Theta(1)$ time and data movement while in sequential mode this is $\mathcal{O}(nz + n)$, where n is the vector size or the sum of the matrix dimensions and nz its number of nonzeroes, regardless of P. Iterating and dereferencing all elements takes $\Theta(nz)$ time and data movement in sequential mode. In parallel mode this bound becomes $\mathcal{O}(nz)$ in the worst-case – good implementations should only incur this bound on adversarially structured data, and otherwise achieve $\Theta(nz/P)$ on average. The *PARALLEL* mode is the default.

A container immediately becomes invalid after the GraphBLAS context has been finalised. A grb::Vector< T > x can, however, be *pinned*:

```
grb::PinnedVector< T > pinned( x, mode );
```

Such a pinned vector persists even after a call to grb::finalize but cannot be an argument to any of the regular GraphBLAS primitives. Possible values for mode are *SEQUENTIAL* and *PARALLEL*; the pinned vector will correspond to the complete vector, respectively, its process-local elements only. This functionality is required when integrating GraphBLAS programs in a larger framework without having to explicitly copy back data every time a GraphBLAS call completes. The creation of a pinned vector in parallel mode shall incur $\Theta(1)$ time and data movement, and shall not make any system calls. Constructing a pinned vector in sequential mode must be done as a collective call, incurs $\mathcal{O}(n)$ time and data movement, and may require memory allocation. Accessing single element indices, values, and mask information of pinned vectors incur $\Theta(1)$ costs in both modes.

2.4. **Operators, Monoids, and Semirings.** Central to GraphBLAS is the explicit use of linear algebraic relations to express computations. A binary operator is a function $T1 \times T2 \to T3$. The grb::operators namespace contains several standard ones listed in Table 2. Note that a small set of operators always produce a boolean value in $B = \{false, true\}$, or may additionally require that its arguments are in B. All operators support compile-time introspection using a type-traits like approach summarised in Table 3. Operators are used by passing them to GraphBLAS primitives as arguments, thus requiring instantiation.

grb::is_operator< *OP* >::value	*true* (for any GraphBLAS operator)
grb::is_associative< *OP* >::value	$f(a, f(b, c)) = f(f(a, b), c)$
grb::is_commutative< *OP* >::value	$f(a, b) = f(b, a)$
grb::is_idempotent< *OP* >::value	$f(a, a) = a$

TABLE 3. Operator traits (left) determine properties of a GraphBLAS templated operator type *OP* < T1, T2, T3 >. Assuming the operator is instantiated with all of $T\{1, 2, 3\}$ equal to the same type T and disregarding any machine precision issues, the returned *value* shall be *true* if the conditions on the right column are met for all $a, b, c \in T$.

grb::identities::zero< T >	0
grb::identities::one< T >	1
grb::identities::negative_infinity< T >	$-\infty$
grb::identities::infinity< T >	∞
grb::identities::logical_true< T >	*true*
grb::identities::logical_false< T >	*false*

TABLE 4. Standard identities available in GraphBLAS (left) and what they represent (right).

A GraphBLAS monoid is formed by combining an associative operator $f : T1 \times T2 \to T3$ with an identity $0 \in T1 \cap T2$ for which $f(a, 0) = a$ for all $a \in T1$ and $f(0, b) = b$ for all $b \in T2$. This additionally requires that each element in *T1* and *T2* can be cast into one in *T3*. A monoid is declared as grb::Monoid< *T1*, *T2*, *T3*, *OP*, *ID* >, where *OP* must be a GraphBLAS operator and *ID* the identity under which *OP* forms a monoid. Table 4 summarises the standard identities GraphBLAS provides. A custom identity must be a non-templated class that publicly exposes a single static constexpr member function *value* that is templated with a single type argument T, which returns the intended identity of the requested type T. For example, grb::identities::zero may be implemented as

```
class zero {
 public:
  template< typename T >
  static constexpr T value() { return static_cast< T >(0) };
};
```

Attempting to form a monoid with a non-operator or a non-associative *OP* will result in a compile-time error, as will declaring a monoid with domains for which the given identity cannot be instantiated. The traits from Table 3 are also defined for monoids. The grb::is_operator< T >::value shall, however, read false for monoids T, while the added trait grb::is_monoid< T >::value reads *true* true if and only if T is a monoid.

A GraphBLAS semiring is formed by combining a monoid with operator $\otimes : T1 \times T2 \to T3$ with a commutative monoid with operator $\oplus : T3 \times T4 \to T4$. The identity of the former is dubbed 'one' (**1**) and that of the latter 'zero' (**0**). The user must ensure that

- $a \otimes (c \oplus d) = (a \otimes c) \oplus (a \otimes d)$ (left-distributivity), and
- $(c \oplus d) \otimes b = (c \otimes b) \oplus (d \otimes b)$ (right-distributivity),

while assuming compatible domains and ignoring limited machine precision. A semiring type is declared via grb::Semiring< T1, T2, T3, T4, OP1, OP2, ID1, ID2 >, where OP1 is the additive operator, OP2 the multiplicative one, and ID$\{1, 2\}$ their respective identities. Such a type is invalid if *OP1* or *OP2* is not associative, if *OP2* is not commutative, or if any of the identities cannot be instantiated in all of the domains T$\{1, 2, 3, 4\}$. Declaring such an invalid semiring shall result in compile-time errors. Additionally, the time trait grb::is_semiring< T >::value exists and only reads *true* if T is a semiring.

Monoids M and semirings S expose public member functions to ease programming. M::getOperator returns the operator underlying the monoid while the templated public member function

```
template< T >
constexpr T M::getIdentity();
```

returns the identity element cast to the given type T. S::getAdditiveMonoid and S::getAdditiveOperator return the underlying additive monoid and its operator, respectively. S::getMultiplicativeMonoid and S::getMultiplicativeOperator return the underlying multiplicative monoid and operator, respectively. These functions return the requested instances by value. The getZero templated public member function corresponds to the getIdentity function of the Semiring's additive monoid while S::getOne corresponds to that of the multiplicative monoid.

Custom operators $\odot$ may be defined as a templated class $C <$ T1, T2, T3 $>$ implementing one or more of the following static member functions:

- C::apply(const T1 &x, const T2 &y, T3 &z); // $z \leftarrow x \odot y$
- C::foldl(T1 &x, const T2 & y); // $x \leftarrow x \odot y$
- C::foldr(const T1 &x, T2 & y); // $y \leftarrow x \odot y$

The class must provide the following static constexpr member values:

- bool C::is_associative,
- bool C::is_commutative, and
- bool C::is_idempotent;

mirroring the traits in Table 3. All above members should be public and set according to the properties of $\odot$. Then, wrapping C using grb::operators::Operator< C > yields a fully GraphBLAS-compatible binary operator, which can also be used to construct custom monoids and semirings.

Instantiating, copying, moving, or destroying a GraphBLAS operator, monoid, or semiring shall incur $\mathcal{O}(1)$ process-local run-time and data movement. The same applies to calling any operator, monoid, or semiring member functions. If operators are implemented without state, an optimising compiler may not produce run-time instantiations; the requested operations are then resolved at compile time.

2.5. **Primitives on scalars and containers.** We divide primitives into scalar operations (level 0), vector operations (level 1), and matrix–vector operations (level 2). While the API of a sparse matrix–sparse matrix multiplication (level-3) is easily similarly defined, our early explorations with one use case in particular indicates that either the GraphBLAS specification needs to be altered for the user to pass information on how containers are related to one another. We hence leave its more precise design for future work, and include preliminary discussions in Section 7.

2.5.1. *level 0.* A GraphBLAS operator f : *T1* $\times$ *T2* $\rightarrow$ *T3* may be applied on input scalars x, y and with output scalar z by

```
grb::apply( z, x, y, f ); // z <- f( x, y )
```

If the argument types do not already match, the inputs x, y will be cast to *T1*, *T2* prior to application, while the output will be cast from *T3* to the type of z after application of f. The operator thus always is applied the types $T\{1, 2, 3\}$ it was parametrised with. The following two primitives apply f in-place under the same casting rules:

```
grb::foldl( x, y, f ); // x <- f( x, y )
grb::foldr( x, y, f ); // y <- f( x, y )
```

These level-0 primitives always return *SUCCESS*, do not make system calls, and incur $\Theta(1)$ time and data movement. These primitives are useful for functions that are parametrised by operators, monoids, and/or semirings.

2.5.2. *level 1.* As with the standard dense BLAS definitions, we define primitives that deal with vectors and scalars to be level-1 operations. Table 5 lists all vector primitives. Some primitives may take either an operator or a monoid, while others require a semiring; see, for example, the grb::foldl and grb::dot, respectively. For primitives that have multiple vectors as input, the sizes of those vectors must be equal or *MISMATCH* will be returned. Unrecoverable internal errors will result in *PANIC*, while if the computation proceeded as intended *SUCCESS* will be returned.

	Arguments			
GraphBLAS primitive	Scalar	Vector	Sp	Element-wise def.
grb::set(x, y, i)	y	x	✓	$x_i = y$ (the given i only)
grb::set(x, y)	y	x	✓	$x_i = y$ (for all i)
grb::set(x, y)		x, y	✓	$x_i = y_i$
grb::clear(x)		x	✓	$x_i = \emptyset$
grb::apply$(z, x, y, \odot)$		z, x, y	✗	$z_i = x_i \odot y_i$
grb::apply$(z, x, y, \odot)$	x	z, y	✗	$z_i = x \odot y_i$
grb::apply$(z, x, y, \odot)$	y	z, x	✗	$z_i = x_i \odot y$
grb::foldl$(x, y, \odot)$		x, y	✗	$x_i = x_i \odot y_i$
grb::foldr$(x, y, \odot)$		x, y	✗	$y_i = x_i \odot y_i$
grb::dot$(z, x, y, \oplus, \otimes)$	z	x, y	✗	$z = z \oplus (x_i \otimes y_i)$
grb::apply(z, x, y, M)		z, x, y	✓	$z_i = x_i \odot y_i$
grb::apply(z, x, y, M)	x	z, y	✓	$z_i = x \odot y_i$
grb::apply(z, x, y, M)	y	z, x	✓	$z_i = x_i \odot y$
grb::foldl(x, y, M)		x, y	✓	$x_i = x_i \odot y_i$
grb::foldl(x, y, M)	y	x	✓	$x_i = x_i \odot y$
grb::foldl(x, y, M)	x	y	✓	$x = x \odot y_i$
grb::foldr(x, y, M)		x, y	✓	$y_i = x_i \odot y_i$
grb::foldr(x, y, M)	x	y	✓	$y_i = x \odot y_i$
grb::foldr(x, y, M)	y	x	✓	$y = x_i \odot y$
grb::add(z, x, y, R)		z, x, y	✓	$z_i = x_i \oplus y_i$
grb::mul(z, x, y, R)		z, x, y	✓	$z_i = x_i \otimes y_i$
grb::muladd(z, a, x, y, R)		z, a, x, y	✓	$z_i = (a_i \otimes x_i) \oplus y_i$
grb::muladd(z, a, x, y, R)	a	z, x, y	✓	$z_i = (a \otimes x_i) \oplus y_i$
grb::dot(z, x, y, R)	z	x, y	✓	$z = z \oplus (x_i \otimes y_i)$

TABLE 5. Level-1 primitives, which arguments are scalars and which are vectors, whether they allow for sparse vectors, and their element-wise definition. Only the first primitive operates on a single index i, all others are applied for each index $0 \leq i < n$ in turn, with n the vector size. The $\odot$ denotes an operator, while M and R denote a monoid and semiring, respectively.

Sparse vectors require interpretation of 'missing' nonzeroes in the container. We opt for a strict interpretation where sparse vectors are only allowed when monoids and semirings are given; an operator-based primitive when given one or more sparse vectors shall hence return *ILLEGAL*.

Table 6 summarises the cost of all level-1 primitives for both sparse and dense vector inputs. These are not exact; we assign costs that are proportional to the big-Theta costs we would reasonably expect. Section 4.1 describes how to cost algorithms that are composed of GraphBLAS primitives, while Section 4.2 considers how well our implementations behave as the performance semantics we here define.

The work and operator application costs of Table 6 are divided by the number of threads T a backend employs. The amount of data movement from memory to CPUs remain unchanged. When $T > 1$ we account for parallel overhead by adding a cost T to work, operator, and data movement. For $P > 1$ we add $\mathcal{O}(P)$ reduction costs for those level-1 primitives with scalar output. Similar to Table 6, we turn this into a simplified cost scheme in terms of work, operator applications, inter-process (as opposed to intra-process) data movement, and synchronisations: $P - 1$ for work, operator applications, and inter-process data movement, and 1 for synchronisation. This cost specification is reasonable when inter-process communications are full duplex and communication costs are proportional to the bottlenecks created at the communication end points– namely, the user processes. This performance specification does *not* imply implementations must use synchronous collectives, nor does it mandate the use of a single-phase reduction; any mechanism that guarantees the indicated cost as an upper bound is acceptable.

Primitive	Work	Ops	Data movement
$\text{set}(x, y, i)$	1	-	1
$\text{set}(x, y)$	n	-	$n + nz_y$
$\text{clear}(x)$	n	-	n
$\text{apply}(z, x, y, \odot/M)$	$\max\{n, nz_x + nz_y\}$	$nz_{x\cap y}$	$2\max\{n, nz_x + nz_y\}$ $+ nz_{x\cup y}$
$\text{foldl}(y, x, \odot/M)$ $\text{foldr}(x, y, \odot/M)$	nz_x	$nz_{x\cap y}$	$2nz_x$
$\text{foldl}(y, \alpha, \odot/M)$ $\text{foldr}(\alpha, y, \odot/M)$ $\text{foldl}(\alpha, y, M)$ $\text{foldr}(y, \alpha, M)$	nz_y	nz_y	nz_y
$\text{add}(z, x, y, R)$	$\max\{n, nz_x + nz_y\}$	$nz_{x\cap y}$	$2\max\{n, nz_x + nz_y\}$ $+ nz_{x\cup y}$
$\text{mul}(z, x, y, R)$	$\min\{nz_x, nz_y\}$	$nz_{x\cap y}$	$2\min\{nz_x, nz_y\} +$ $nz_{x\cap y}$
$\text{dot}(z, x, y, (\oplus, \otimes))$	n	$2n$	$2n$
$\text{dot}(z, x, y, R)$	$\min\{nz_x, nz_y\}$	$2 \cdot nz_{x\cap y}$	$2\min\{nz_x, nz_y\}$
$\text{muladd}(z, a, x, y, R)$	$\min\{nz_a, nz_x\}$ $+ nz_y$	$nz_{a\cap x}$ $+ nz_{y\cup(a\cap x)}$	$2\min\{nz_a, nz_x\}$ $+ nz_y + nz_{a\cup(x\cap y)}$

TABLE 6. The costs of applying level-1 primitives in work (left), the number operator applications (middle), and data movement (right). In case of scalar a, x or y, the corresponding nz should be interpreted as 1. For vectors a, x, and y their lengths and number of nonzeroes are denoted by n and nz, respectively.

Often, several level-1 operations occur in sequence; e.g.,

```
grb::Vector< double > z(n), w(n), x(n), y(n);
grb::operators::add< double, double, double > addR;
grb::operators::sub< double, double, double > subR;
grb::set( x, 2.0 ); grb::set( y, 1.0 ); // dense vectors of twos and ones
grb::apply( z, x, y, addR );            // z = x + y; a vector of threes
grb::apply( w, x, y, subR );            // w = x - y; a vector of ones
```

According to the cost semantics attached to grb::apply, the above code would incur $6n$ data accesses– the optimal, however, is $4n$. To attain this optimal bound while retaining clear cost semantics we define the grb::eWiseLambda primitive. This primitive takes an arbitrary number of equally-sized vectors and one lambda function as arguments and executes the latter element-wise:

```
grb::eWiseLambda( [&x,&y,&z,&w,&addR,&subR] (size_t i) {
    grb::apply( z[ i ], x[ i ], y[ i ], addR );
    grb::apply( w[ i ], x[ i ], y[ i ], subR );
}, x, y, z, w );
```

The grb::eWiseLambda incurs a total work and data movement cost proportional to nz_x, multiplied by the number of vector arguments. The number of operator applications is nz_x multiplied with the number of operators applied within the lambda function. The above may replace the two grb::apply primitives in the preceding example without changing their functional semantics, improving their cost only.

In case of sparse input vectors, the eWiseLambda follows the nonzero structure of the first given vector (x, in the above example) and requires that all other vectors have a matching nonzero entry for every nonzero in that first vector– otherwise, *ILLEGAL* will be returned. As a consequence this function cannot create new nonzero entries.

2.5.3. *level 2.* We define two primitives for sparse matrix–vector (SpMV) multiplication:

```
grb::mxv( y, A, x, R ); // y = Ax
grb::vxm( y, x, A, R ); // y = xA
```

	$nz_{mask} < nz_x$	$nz_{mask} \geq nz_x$
Work	$nz_{mask} + \sum_{i:mask_i=true} nz_{A_{i,:}}$	$nz_x + \sum_{j:x_j \neq 0} nz_{A_{:,j}}$
Data movement		
- obligatory	$nz_{mask} + \sum_{i:mask_i=true} nz_{A_{i,:}}$	$nz_x + \sum_{j:x_j \neq 0} nz_{A_{:,j}}$
- best case	$nz_x + nz_{y \cap mask}$	$nz_x + nz_{y \cap mask}$
- worst case	$\sum_{i:mask_i=true} nz_{A_{i,:}}$	$\sum_{j:x_j \neq 0} nz_{A_{:,j}}$
Operator applications	$2\lvert\{i,j \mid A_{ij} \neq 0 \wedge mask_i = true \wedge x_j \neq 0\}\rvert$	

TABLE 7. The costs of applying the level-2 grb::mxv(y, m, A, x) in work, data movement, and number of operator applications. The former two depend on the sparsity of the *mask* versus that of the input vector x. The number of operator applications does not depend on this distinction. The data movement is subdivided into three rows: the amount of data that will be moved no matter what (obligatory), to which either the best- or worst-case data movement must be added. In case no *mask* was given then nz_{mask} defaults to m while $mask_i = true$ everywhere. Substituting A^T for A and interchanging m and n yields the cost of grb::vxm(y, m, x, A).

The input vector x may be sparse or dense, while the final result y may be sparse or dense as well. The input matrix A is assumed sparse always. Multiplication occurs using the supplied semiring R such that for all nonzeroes $a_{ij} \in A$

$$y_i \leftarrow (x_j \otimes a_{ij}) \oplus y_i, \tag{1}$$

working in-place on output vector elements y_i for the mxv. Similarly, for the vxm:

$$y_i \leftarrow (x_i \otimes a_{ij}) \oplus y_j, \tag{2}$$

for all nonzeroes $a_{ij} \in A$. The order of operator applications is implementation-defined. Modulo round-off differences and casting behaviour should the semiring domains differ this order should not matter; otherwise the given $\oplus, \otimes$ do not form a semiring. All elements of y that may have existed prior to the BLAS-2 call shall be ignored. Both primitives return *MISMATCH* whenever the size of one of the supplied vectors or matrices do not match the corresponding dimension of A, and return *ILLEGAL* if the output container is the same as one of the input containers.

As far as the authors know, there are no methods in practical use that achieve asymptotically better bounds for sparse matrix storage requirements than standard Compressed Row Storage (CRS) or its column-wise variant (CCS) [BFF$^+$09, YR13, YLZZ14, KHW$^+$14, LV15, SC19]; different formats may be in use but do not improve significantly on the bounds of CRS and CCS. Hence if the input vector x is dense and there is no masking, we assume level-2 primitives incur $\Theta(2nz_A)$ work and a lower and upper bound on data movement of $\Omega(nz_A + nz_x + nz_y)$ and $\mathcal{O}(3nz_A)$, respectively. We associate a cost of $nz_A + \min\{m, n\}$ to both the grb::vxm and grb::mxv accordingly. These costs are summarised in Table 7 and clarified further in Section 2.6 to account for masking. All costs except data movement are divided by the number of threads T a backend supports, while all costs are increased with T to account for parallelisation overhead. Unlike level-1 primitives, backends that support $P > 1$ have level-2 primitives with implementation-defined inter-process costings. None of the level-2 primitives shall make system calls and none shall ever allocate or free dynamic memory. Implementations hence must pre-allocate any necessary buffers on initialisation, on calls to grb::resize, and/or on container construction.

2.6. **Masks and accumulators.** All level-1 and level-2 primitives that have vector output arguments, with the exception of the element-wise grb::set, can take masks as an additional argument. A *mask* is a boolean vector with size equal to the output vector y and should be given as an argument after y. Passing a mask will modify the primitive to only compute those elements y_i for which $mask_i$ evaluates *true*; otherwise y_i remains untouched. For example, the grb::mxv primitive has the following overloaded form:

```
grb::mxv( y, mask, A, x, R );
```

The *mask* may never refer to the same container as y or *ILLEGAL* will be returned.

The use of masks has significant impacts the costing of level-1 primitives with vector outputs. Assuming y indicates the output vector and n its size, masking replaces n with nz_{mask} and nz_y with $\min\{nz_y, nz_{mask}\}$

Descriptor	Effect
transpose	Transposes the input matrix
invert_mask	Negates the mask
structural_mask	Consider the nonzero structure of the mask, not its values
dense	Assume all vector arguments are dense, disable checks for sparsity
in_place	The accumulator is set to the given (additive) operator
no_casting	Disallow casting input or output arguments

TABLE 8. All descriptors residing in the grb::descriptors namespace and their effects.

in Table 6. The number of operator applications and data movement costs have their nz counts intersected with the sparsity pattern of the mask, translating, for example, $nz_{x \cap y}$ to $nz_{x \cap y \cap mask}$.

For the level-2 grb::mxv, there are two options: either the mask is 'sparse enough' to incur a total work and data movement cost of $\mathcal{O}(nz_{mask} + \sum_{i:mask_i=true} nz_{A_{i,:}})$, or the input vector is 'sparser' in which case the total work and data movement is $\mathcal{O}(nz_x + \sum_{j:x_j \neq 0} nz_{A_{:,j}})$. In both cases, the total number of operator applications is $2|\{i, j \mid A_{ij} \neq 0 \wedge mask_i \wedge x_j \neq 0\}|$. Since these computations are expected to be memory-bound on all contemporary hardware, we specify that implementations should achieve $\mathcal{O}(\min\{nz_{mask} + \sum_{i:mask_i=true} nz_{A_{i,:}}, nz_x + \sum_{j:x_j \neq 0} nz_{A_{:,j}}\})$ work and data movement. The costs of grb::vxm are similarly modified but with the mask operating on the columns of A instead of its rows, and the input vector operating on the rows of A instead of its columns. Table 7 summarises our level-2 performance semantics.

The second modification to output may be given via an *accumulator*. Any binary GraphBLAS operator may act as such an accumulator. When given, old values in the output container are not overwritten with a new result; instead, they are combined according to the given operator with the existing value as its left-hand argument and the new result as its right-hand argument. The accumulator output will then be stored.

For level-1 primitives on vectors of length n, the use of an accumulator only adds to the operator application costs: this increases with n. The work and data movement bounds remain unmodified. For level-2 primitives, the use of accumulators adds $\mathcal{O}(n)$ data movement and work, as well as n operator applications to account for any buffering of the output elements that an implementation may require. There, n is the length of the output vector in case of an unmasked operation or $\min\{n, nz_{mask}\}$ otherwise.

2.7. **Descriptors.** Descriptors affect either functional semantics, performance, or compile-time behaviour of a GraphBLAS primitive. Every primitive may be given a set of descriptors as a function template parameter. All descriptors are summarised in Table 8. We follow with an example of a descriptor that affects semantics, performance, or compile-time behaviour:

- The grb::descriptors::transpose indicates the first matrix argument to the given primitive should be interpreted as its transpose, for example to have the grb::vxm compute $y = xA^T$ via `grb::vxm< grb::descriptors::transpose >( y, x, A, R );`.
- The dense descriptor allows a GraphBLAS implementation to remove any run-time checks on sparsity of vector arguments, and emits simpler code by omitting all logic associated to handling sparsity in vectors.
- The no_casting descriptor disallows any casting prior to the application of a GraphBLAS operator. It helps ensure the correct implementation of GraphBLAS programs by, for example, rejecting the compilation of a program where a semiring defined on doubles is applied on integer GraphBLAS containers.

Descriptors can be combined using the standard C++ binary-or operator. For example, computing $y = A^T x$ where y and x are guaranteed dense can be expressed via

```
grb::mxv< grb::descriptors::transpose | grb::descriptors::dense >(
    y, A, x, R
);
```

Error codes such as *MISMATCH* shall take into account descriptors such as transpose. Performance bounds change according to how input is modified by descriptors. For example, bounds depending on the number of *true* elements in a mask will be replaced by the number of *false* elements instead if the invert_mask descriptor is given, while bounds depending on the number of rows of a matrix will depend on the number of columns instead if the transpose descriptor is given.

3. Implementation

We implemented three backends for our C++ GraphBLAS API: a vectorising, an shared-memory parallel, and a distributed-memory parallel one. The latter is the only backend we present here that supports more than one user process. It must be configured using one "sub-backend" that provides the implementation for process-local computations. Choosing the shared-memory backend for this naturally results in a fully hybrid shared- and distributed-memory GraphBLAS.

3.1. Vectorising backend.

For storing GraphBLAS vectors of size n and type T we allocate three arrays:

- a *value array* of length n and type T,
- an *assigned array* of length n and boolean type, and
- a *stack* of length n and a configurable unsigned integer (uint) type.

As the specification requires, allocation occurs on construction of the vector.

The vector maintains a count of the current number of nonzeroes, nz. If for a BLAS-1 primitive all involved vectors are dense, i.e., nz equals n, the primitive falls back to fully vectorised code using only the value arrays. If a vector is sparse, a level-1 or level-2 primitive requires either fast query access or fast iteration access. For the former, i.e., to answer whether an ith element is nonzero, we query the *assigned* array. The latter, i.e., to iterate over all nonzero indices, we employ the *stack* instead. Adding a nonzero to a vector requires updating the assigned array and pushing the corresponding index on the stack, as described by Algorithm 1. Clearing a vector requires resetting the assigned array and thus is a linear-time operation. This data structure only supports removing *all* nonzeroes in $\Theta(n)$ costs, and does not support removing single nonzeroes.

Algorithm 1 Adding a nonzero to a (possibly sparse) vector

Input: x, a vector container of size n
i, an index smaller than n
$\oplus$, an additive operator
val, a value to add to the ith element of x
Output: x, the updated input vector

1: **if** x is dense **or** $x.assigned_i$ **then**
2: $x.values_i \leftarrow x.values_i \oplus val$
3: **else**
4: Set $x.assigned_i$ to *true*
5: Add i to $x.stack$
6: Increment $x.nz$
7: Set $x.values_i$ to val
8: **return** x

A matrix is stored using Gustafson's data structure, i.e., storing the matrix twice, once using compressed row-major storage (CRS) and another using compressed column-major storage (CCS). While doubling the memory requirement this choice allows for efficient operations for both the grb::vxm and grb::mxv primitives. This data structure allows fast access to both matrix rows and columns, iterating nonzeroes contained therein at $\Theta(1)$ amortised costs.

All level-1 primitives except eWiseLambda translate to fully vectorised code on all tested compilers[2], designed to achieve performance equal or exceeding that of compiler-optimised manually-coded for-loops. To achieve this, our implementation of level-1 primitives first loads all elements into buffers that fit a configurable SIMD size, and then applies the requested operations within those buffers. Once ready, output buffer elements are written back to main memory before processing the next batch. The buffer size automatically adjust to the operator data types through template meta-programming.

Instead of detailing the implementation of every primitive, we focus on the grb::mxv only as it captures all of the uses of both the vector and matrix data structure. For readability we omit all logic regarding descriptors. Those are implemented using if-branches that are collapsed at compile time by virtue of being passed as template arguments. There are four main variants of performing the grb::mxv, made up by the Cartesian product of the following two choices: gather vs. scatter and row-major vs. column-major.

Let $k = \min\{nz_{mask}, m\}$ when a mask is given or m otherwise and $l = \min\{nz_x, n\}$, where m, n are the row, resp., column dimensions of A in $y = Ax$. The row-major choice is the data structure of choice

[2]As tested with Clang/LLVM version 6, ICC 2018.2, GCC 4.8.3, and GCC 8.3.1 on Intel architectures.

whenever $k \leq l$, while the column-major data structure is preferred otherwise; Algorithms 2 and 3 illustrate both. Using either the row- or column-major data structure in this fashion is known in graph processing as *pull* or *push*, respectively, while automating the choice is also known as *direction optimisation* [BAP12].

Algorithm 2 Implementation of grb::mxv ($y = Ax$) using row-major CRS

```
                 y, mask, x, grb::Vector
      Input:           A, grb::Matrix
                       R, grb::Semiring< ..., ⊕, ⊗, 0, 1 >
     Output:   y ← y ⊕ Ax under semiring R, grb::Vector
   Requires:   Algorithm 4 as gather_spmv
1: if y is masked then
2:   for all nonzero indices i in mask do
3:     if mask_i evaluates true then
4:       gather_spmv(y_i, i, A.CRS, x, R)
5: else
6:   for i = 0 to A.m − 1 do
7:     gather_spmv(y_i, i, A.CRS, x, R)
8: return y
```

Algorithm 3 Implementation of grb::mxv ($y = Ax$) using column major CCS

```
                 y, mask, x, grb::Vector
      Input:           A, grb::Matrix
                       R, grb::Semiring< ..., ⊕, ⊗, 0, 1 >
     Output:   y ← y ⊕ Ax under semiring R, grb::Vector
   Requires:   Algorithm 5 as scatter_spmv
1: if x is sparse then
2:   for every nonzero index j in x do
3:     scatter_spmv(y, mask, A.CCS, x_j, R)
4: else
5:   for j = 0 to A.n − 1 do
6:     scatter_spmv(y, mask, A.CCS, x_j, R)
7: return y
```

Each variant is expressed using the gather_spmv and scatter_spmv kernels as an inner loop. The former takes a row index i, multiplies each nonzero a_{ij} on the ith row of A with their corresponding x_j, and accumulates the result in y_i; it touches all elements of x and only a single element of y. By contrast, the scatter_spmv takes a column index j, multiplies each nonzero a_{ij} on the jth column of A with x_j, and adds each contribution to their corresponding y_i; it touches (and adds to) each element of y and only a single element from x. Algorithms 4 and 5 express these operations using the vector data structure described earlier.

To implement the grb::vxm ($y = x^T A$) the same algorithmic components can be used– a row-major grb::vxm would require a scattering inner kernel (Alg. 5) and is preferable over a gathering column-major variant whenever $\min\{nz(x), m\} < \min\{nz(mask), n\}$.Both the gather and scatter inner kernels (Algs. 4, 5) indeed require $\Theta(1)$ access and updates to the sparsity structure of the output vector, while both row- and column-major variants (Algs. 2, 3) require $\Theta(1)$ access time for iterating over nonzeroes of sparse input vectors and masks.

The use of an accumulator with a scatter variant requires a completely buffered output. Since only container constructors are allowed to allocate memory, allocating a grb::Vector will check if a centralised buffer has enough space to hold a copy of itself; if not, a reallocation of the internal buffer to the required size takes place.

In the vectorising backend, the *SEQUENTIAL* and *PARALLEL* I/O modes are equivalent since there is but one user processes. The parser required to ingest data of different formats is beyond the scope of a GraphBLAS implementation which interfaces with parsers only through the iterators such parsers should expose. In particular, implementations will not deal with double buffering and other techniques commonly

Algorithm 4 Processes a single output element during SpMV (gather).

Input: y_i, scalar
x, grb::Vector
A, grb::Matrix
R, grb::Semiring< $\ldots, \oplus, \otimes, \mathbf{0}, \mathbf{1}$ >
Output: $y_i \leftarrow y_i \oplus Ax$ under semiring R

1: **if** not $y.assigned_i$ **then**
2: Set $y.assigned_i$ to *true*
3: add i to $y.stack$
4: increment $y.nz$
5: set $y.value_i = \mathbf{0}$
6: **for** every nonzero a_{ij} on row i of A **do**
7: $y_i \leftarrow y_i \oplus (a_{ij} \otimes x_j)$
8: **return** y_i

Algorithm 5 Processes a single input vector element during SpMV (scatter).

Input: x_j, scalar
$y, mask$, grb::Vector
A, grb::Matrix
R, grb::Semiring< $\ldots, \oplus, \otimes, \mathbf{0}, \mathbf{1}$ >
Output: $y \leftarrow y \oplus Ax_j e_n$ under semiring R
Requires: Algorithm 1 as *add_nonzero*

1: **for** every nonzero a_{ij} on the jth column of A **do**
2: **if** $mask.n > 0$ and $mask_i$ evaluates *false* **then**
3: **continue**
4: Set *mul* to $a_{ij} \otimes x_j$
5: *add_nonzero*$(y, i, \oplus, mul)$
6: **return** y

required to achieve good I/O performance– these are solely the domain of the parser. We hence limit our description to how our implementation uses forward iterators to ingest vectors and matrices.

For vectors, our implementation streams through the input iterators once while building up on the fly the corresponding stack and array that encode the nonzero structure. For matrices, we stream through the input coordinate set twice– once to count the number of nonzeroes per row and column, and a second pass to ingest the nonzeroes into the dual CRS and CCS structures. The values iterators are passed through once during the last phase. This is equivalent to a counting sort to bring the input nonzeroes in row- and column-major order. We do not sort the nonzeroes within individual rows and columns of the CRS and CCS structures, respectively, and thus achieve matrix ingestion at linear $\Theta(nz)$ cost.

Any out-of-memory conditions encountered in constructors raise the corresponding C++ exception, while those encountered within GraphBLAS primitives the corresponding error code is returned instead. When the no_casting descriptor is provided, the C++11 static_assert ensures that domains of containers match those of the provided operator, monoid, or semirings; if not, a compile-time error occurs.

Checks on matching sizes of vectors and matrices occur at run-time. Detected errors result in GraphBLAS primitives returning MISMATCH. If sparse containers are provided to primitives that can only handle dense ones, ILLEGAL is returned. Such run-time checks are done prior to any instruction that modifies any container arguments.

3.2. **Shared-memory parallel backend.** The shared-memory parallel backend is a modification of the vectorising backend. When allocating memory, this backend distinguishes between *local* memory areas that are guaranteed to be touched by a single thread versus *shared* memory areas that may be touched by multiple threads. On initialisation of the GraphBLAS context (grb::initialize), the calling process uses non-portable POSIX Threads extensions by GNU to retrieve the current UNIX process mask. If the mask is full, we assume no other GraphBLAS processes are active on the same machine: all shared memory regions shall hence be allocated in an interleaved fashion by use of the libnuma library. If, however, the mask is sparse, we assume multiple GraphBLAS processes are active on the same machine that we furthermore assume are pinned to individual NUMA domains. All shared memory regions shall then be

allocated using the local allocation policy, again enforced via libnuma. This ensures good performance on NUMA architectures, regardless of whether one process per node or one process per NUMA domain is employed. The vector and matrix data structures mentioned in the previous subsection are all examples of shared memory regions.

The default buffer size for the shared-memory backend is TL bytes, with T the number of OpenMP threads when starting a parallel region and L the cache line size in bytes. This buffer is an example of a local memory region. Buffer sizes may only increase whenever a container is constructed or grb::resize is called. Primitives expressed using for-loops in the vectorising backend are parallelised using the standard OpenMP parallel-for pragma. Dense access patterns such as looping over all vector elements or over all rows of a CRS structure employ a static schedule. Sparse access patterns such as looping over nonzero elements or over masked rows of a CRS structure use a dynamic schedule with chunk size L.

More intricate are updates to the sparsity structure of vectors on lines 2–5 of Algorithm 1 and the lines 4–7 of Algorithm 1. Assuming updates are processed in parallel, we first observe that 1) setting the assigned and value arrays for different indices never results in data races for vector operations and for the gather variant of the SpMV multiplication, while 2) the nonzero count may only ever increase and never decrease. Thus when each thread maintains a local stack and a local new nonzero count, updates can proceed in parallel. If the number of nonzeroes at thread t is nz_t, we may perform a prefix-sum on the nz_t to obtain the offsets for copying local stack contents into the global stack location. After this collaborative prefix sum, each thread then copies its local stack into the global data structure without write conflicts and with only a maximum of $\mathcal{O}(T)$ cache lines subject to false sharing. The total cost of this operation is $\Theta(T + \max_t nz_t)$, where the cost of both the prefix sum and the allreduce on the nz_t is assumed asymptotically equal or smaller. This may lead to imbalanced updates, however, if $\max_t nz_t$ is significantly higher than $1/T \sum_t nz_t$. An alternative which avoids such imbalances instead parallelises the copying of local stacks into the global stack, at a cost of $\Theta(\sum_t \lceil nz_t/T \rceil)$. This variant, however, should not be use over multiple NUMA nodes as nearly all memory traffic would be limited by the throughput of a single memory controller.

Algorithm 6 presents a variant that combines the NUMA-friendly but imbalance intolerant sparsity updates with the NUMA unfriendly but balance tolerant approach. It incurs non-local memory accesses only when an imbalance in new nonzeroes presents itself. The local stacks are a partitioning of the internal buffer. The maximum size of the local stacks are bounded by the vector length n, but to retain scalability we instead allow for local stacks of maximum size n/T. The internal buffer is resized accordingly on the first instantiation of a vector of size n. If during a sparsity update at some index i the maximum local stack size is reached, then the local stacks will be processed early, consuming all nonzeroes added to the thread-local stacks and resetting them. We then continue the computation at index $i + 1$. Although we did not perform an in-depth performance study, we believe this mid-way variant suffices and employ it within this backend.

Parallel sparsity updates of the output vector cannot be applied for the scattering variant of the SpMV multiplication (Alg. 5): there, multiple threads could concurrently update the sparsity of any element of the output vector, thus causing data races. We hence do not parallelise the scattering SpMV multiplication. Instead, when selecting which variant to use, the cost of the scatter variant is multiplied by T to account for the overhead of $T - 1$ idle threads. Exploiting sparsity on either the input vector or the output mask may hence still lead the implementation to select the sequential scattering SpMV multiplication over a parallel gathering one; Section 4.2.4 confirms the practical benefit of this strategy.

As with the vectorising backend, there is no difference between the *SEQUENTIAL* and *PARALLEL* I/O modes; even though there are T threads active, there is but one user process. The routines for data ingestion follow trivially from the preceding discussion, or can be considered an OpenMP counterpart of the ingestion algorithm used by the distributed-memory parallel backend described in the next section; we therefore omit its precise description here. Data extraction occurs via iterators in a sequential fashion. Error handling is unchanged from the vectorising backend, except that multiple threads may concurrently run into different errors. In such cases, this backend only returns one of those and does not specify which. Most errors, however, such as dimension mismatches, are unambiguous and handled before any OpenMP parallel section is created.

3.3. **Distributed-memory parallel backend.** This backend is the only we describe that allows multiple user processes. It splits algebraic operations into process-local operations with inter-process communication between them, as required. For the former we rely on any of the previously described single user-process backends that must be selected at compile time. For inter-process communication we make use of the Lightweight Parallel Foundations (LPF) [SY19], which promises predictable bounds on communication performance.

Algorithm 6 Parallel buffered updates to a vector nonzero structure.

Preliminaries:

t, my thread ID
T, the total number of threads
L, the cache line size, in bytes
offset, $\max\{1, L/\mathit{sizeof}(\mathit{size\_t})\}$
x, the vector to add new nonzeroes to
$\oplus$, the additive operator

Buffer requirements:

1: Let $\mathit{new\_nz}$ be an array of $(T+1) \cdot \mathit{offset}$ size_t elements initialised to 0
2: T arrays $\mathit{local\_stacks}$ of size $\lceil n/T \rceil$ size_t elements

During a parallel-for driven call to a gather-kernel (Alg. 4) or a scatter one (Alg. 5), thread t replaces vector sparsity updates for new nonzero values (Alg. 1) with:

1: Set $x.\mathit{assigned}_i$ to *true*
2: Set $x.\mathit{values}_i$ to *val*
3: Increment $\mathit{new\_nz}_{t \cdot \mathit{offset}}$ {Padding with *offset* avoids false sharing}
4: Add i to $\mathit{local\_stack}_s$

When all threads are done or when any thread has a full local buffer, execute sequentially:

1: **for** $s = 0$ to $T - 1$ **do**
2: Let $\mathit{new\_nz}_{(s+1)\cdot\mathit{offset}} \leftarrow \mathit{new\_nz}_{(s+1)\cdot\mathit{offset}} \oplus \mathit{new\_nz}_{s\cdot\mathit{offset}}$ {Non-local accesses}
3: **parallel-for** $i = 0$ to $\mathit{new\_nz}_{T-1}$ (static schedule) **do** {This executes in parallel}
4: Determine maximum k for which $i < \mathit{new\_nz}_k$
5: Let $j = i - \mathit{new\_nz}_k$
6: Copy $\mathit{local\_stack}_{kj}$ to $x.\mathit{stack}_{nz+i}$ {Non-local accesses when $k \neq t$}
7: Add $\mathit{new\_nz}_{\mathit{offset}(T-1)}$ to $x.nz$

Vectors and matrices are distributed across user processes using a 1D block-cyclic distribution, meaning that the ith element of a vector or the ith row of a matrix is distributed to user process $s = \pi(i) = \lfloor i/b \rfloor \bmod P$. On that user process, the *global index* i is translated to a local index $i_{\text{local}} = \nu(i) = \lfloor i/(Pb) \rfloor \cdot b + (i \bmod b)$. The parameter b is configurable, in our case set to the cache line size of the Intel-based architectures experimented with in Section 4.2; i.e., $b = 64$. This choice ensures a vector of chars, which all have element size of one byte, will never break up a single cache line across different user processors. Internally, we map this distribution to a row-wise 1D block distribution whenever data is ingested into GraphBLAS containers. This simplifies our implementation. Indices are corrected whenever the user extracts data from containers.

The cyclic nature of the distribution naturally balances vectors and matrices with skewed nonzero distributions, though adversarial structures are, of course, constructable. Alternatives to this block-cyclic distribution may be as simple as greedily assigning consecutive index ranges that achieve load balance, or more advanced methods that require explicit sparse matrix partitioning, discussed as future work in Section 7.

Level-1 primitives. While level-0 primitives require no modification, level-1 operations must be passed through to the single-process backend. Purely element-wise primitives do not require post-processing, while level-1 primitives that produce scalar outputs reductions must perform reductions over all user processes. Such post-processing is well-understood in the form of collective operations, described very well by, for example, Chan et al. [CHPvdG07]. These are supplied for most parallel communication frameworks, which GraphBLAS implementations are encouraged to rely on. The specification demands this backend defines a costing in terms of work and operator application overhead, inter-process data movement, and synchronisations. The specification attaches $P - 1$ work, data movement, and operator applications plus 1 synchronisation costs to all level-1 primitives with scalar output. This implies our implementation may only employ tree-based reductions if it can guarantee this would out-perform a single-round reduction.

Level-2 primitives. The level-2 operations similarly can rely on the single-process backend for process-local computations; however, depending on the operation, input vectors may require replication while output vectors may require reduction. To illustrate, we describe both the vxm and mxv operations.

The vxm operation computes $y = xA$ with A in a row-wise 1D distribution. A process requires its local part of the input vector x and its local part of the input matrix A to generate a local output vector y_s of size m via a process-local vxm. In a post-processing stage, each process s then obtains its local part of $\sum_{k=0}^{P-1} y_s$ using a two-phase reduction. The mxv computes $y = Ax$ with A in the same distribution. Each process requires the complete input vector to be available before a process-local mxv can proceed. Thus, each of the P user processes first broadcast their local part of x to each of the other $P - 1$ processes, and then perform the sequential mxv. This immediately completes the computation since each process computes exactly the local part of the output vector it is responsible for.

Synchronising multiple contributions to the same output vector elements is known as *fan-in*, while sending input vector elements to sibling processes who require them is known as *fan-out*. Different from previous work on high-performance sparse matrix–vector multiplications, the fan-in and fan-out stages may occur over sparse instead of dense vectors, resulting in a trade-off between communication and computation. There are three variants of performing the fan-in:

(1) P reductions of P partial *value* arrays of size m/P, only in the case of dense process-local output vectors y_s;
(2) P allgather collective operations that synchronise both the *assigned* and *values* array of each of the y_s, followed by a masked reduction;
(3) sort the indices in the local *stack* of y_s according to π, copy the corresponding *values* in a buffer in matching order, then perform $2P$ alltoalls so that each process receives her own local index and values pairs for local reduction.

The first variant is only employed for dense vectors, and is preferable over the other two variants because it does not communicate the vector nonzero structure. For sparse vectors the second variant takes a total of $\Theta(m)$ communication and process-local data movement, for which the third variant takes $\Theta(nz)$ instead. When synchronising the *assigned* array we work with boolean elements at $\lceil w_{\text{boolean}} \rceil$ byte per entry, while for the stack-based synchronisation elements take w_{indices} byte instead. We thus choose variant two whenever $w_{\text{boolean}} m < w_{\text{indices}} nz$, and variant three otherwise.

For fan-out, the input vector x needs synchronisation. The same three variants as for fan-in are possible, with the difference that no reduction of values is ever required:

(1) if all process-local parts of x are dense, an allgather on the process-local *values* array suffices;
(2) similarly, an allgather on both *values* and *assigned* would synchronise the sparsity pattern and can handle sparse vectors; or
(3) sort the local *stack*, copy the corresponding nonzeroes in a local buffer in matching order, then perform an alltoall on both the sorted *stack* and buffered values.

We dub the fan-in operation *combine* while the fan-out is dubbed *synchronise*, summarised in Algorithms 7 and 8, respectively. These two algorithms, in addition to delegating process-local computations to the single-process backend, are sufficient to implement all our matrix–vector operations under any combination of descriptors on distributed-memory systems[3].

To the grb:vxm we assign $nz_y + P - 1$ work, nz_y operator applications, $nz_y + P - 1$ inter- *and* intra-process data movement, and one synchronisation. These add to the costs in Table 7 for $\lceil m/P \rceil \times n$ matrices and match the array-based and the dense combine in Alg. 7 for which $nz_y = m$. For the grb::mxv, we similarly assign a cost of nz_x work, inter-, and intra-process data movement. Also here we add one synchronisation cost, and add process-local mxv costs for $\lceil m/P \rceil \times n$ matrices. We remind that the specification demands we specify these implementation-defined costs: they only apply to this specific backend. Section 7 discusses other potential distributed-memory parallel backends and sketches an automatic selection mechanism for balancing trade-offs that having multiple backends may give rise to.

Parallel IO. Since in this distributed-memory P may be larger than 1 a sequential IO mode may behave differently from a parallel one. Consider ingesting a collection of nonzeroes given by start and end iterator pairs. If the collection contents are equal at each process, the sequential mode should be selected. If the collection at each process is disjoint while their union represents the complete input set, the parallel mode should be selected. Using the sequential mode incurs a parallel overhead of $\Theta(Pnz)$, with nz the container input size. In parallel mode, the overhead is limited to redistributing the input between

[3]Though not presented here, these two algorithms in fact also suffice for implementing a distributed-memory parallel backend based on 2D Cartesian distributions.

Algorithm 7 Combine P global vectors into P local vectors.

Input: s, P, the local process ID and the total number of user processes
$y_s.global$, an output vector of length m
m_s, the size of $y_s.local$
$\oplus$, the operator under which to reduce output elements
Output: $y_s.local$, the local version of $\oplus_{i=0}^{P-1} y_s$
Requires: Algorithm 1 as *add_nonzero*

Buffer requirements: $P\lceil \frac{w_{\text{boolean}}}{w_{\text{indices}}} m/P \rceil (2w_{\text{values}} + w_{\text{indices}}) + 4Pw_{\text{size\_t}}$ bytes

1: Set nz_s to $y_s.global.nz$
2: Get $max\_nz = \max_k nz_k$ and $min\_nz = \min_k nz_k$ {two allreduces}
3: Get $offset_s = \sum_{k=0}^{s-1} m_k$ {prefix-sum}
4: **if** $min\_nz = m$ **then**
5: **for** $i = 0$ to $m_s - 1$ **do**
6: $y_s.local.values_i = y_s.global.values_{offset_s+i}$
7: **for** $k = 0$ to $P - 1$ excluding $k = s$ **do**
8: $y_s.local.values_i \leftarrow y_s.local.values_i \oplus y_t.global.values_{offset_s+i}$ {P reductions using $\oplus$}
9: Ensure $y_s.local.assigned = true$ everywhere
10: Set $y_s.local.nz = m_s$
11: **else if** $\lceil w_{\text{boolean}} m \rceil < w_{\text{indices}} max\_nz$ **then**
12: Set $y_s.local.nz$ to 0
13: **for** $k = 0$ to $P - 1$ excluding $k = s$ **do**
14: Get m_s elements from $y_k.global.\{assigned, values\}$ at $offset_s$ {two allgathers}
15: **for** $i = 0$ to $m_s - 1$ **do**
16: Set $y_s.local.\{assigned, values\}_i$ according to $y_s.global.\{assigned, values\}_{offset_s+i}$
17: **if** $y_s.local.assigned_i$ **then**
18: Increment $y_s.local.nz$
19: **for** $k = 0$ to $P - 1$ excluding $k = s$ **do**
20: **if** $y_k.global.assigned_{offset_s+i}$ **then**
21: *add_nonzero*($y_s.local$, i, $\oplus$, $y_k.global.values_{offset_s+i}$)
22: **else**
23: Clear $y_s.local$ and sort $y_s.global.stack$ according to π {process-local counting sort}
24: Let nz_k^s be the number of elements in $y_s.global.stack$ for which π maps to k
25: Get nz_s^k, $remote_k = \sum_{t=0}^{s-1} nz_t^k$, and $local_k = \sum_{t=0}^{k-1} nz_s^t$, $\forall k$ {alltoall and two prefix sums}
26: Retrieve buffers out_s and $inval_s$ of $y_s.global.nz$ and $remote_P$ values, respectively
27: Retrieve buffer $instk_s$ of $remote_P$ indices
28: **for** $i = 0$ to $y_s.global.nz - 1$ **do**
29: **if** π_i equals s **then**
30: *add_nonzero*($y_s.local$, $y_s.global.stack_i - offset_s$, $\oplus$, $y_s.global.values_i$)
31: **else**
32: Set $out_{s,i}$ to $y_s.global.values_{y_s.global.stack_i}$
33: **for** $k = 0$ to $P - 1$ excluding $k = s$ **do**
34: Get nz_s^k elements from $y_k.global.stack$ at offset $remote_k$ into $instk_s$ at offset $local_k$
35: Get nz_s^k elements from out_t at offset $remote_k$ into $inval_s$ at offset $local_k$ {two alltoallvs}
36: **for** $k = 0$ to $P - 1$ **do**
37: **for** $i = 0$ to $nz_s^k - 1$ **do**
38: *add_nonzero*($y_s.local$, $y_k.global.stack_{remote_k+i} - offset_k$, $\oplus$, $inval_{s,remote_k+i}$)
39: **return** $y_s.local$

processes. Assuming input elements are initially distributed uniformly, the expected overhead is $\Theta(nz)$. The worst-case overhead, however, is $\mathcal{O}(Pnz)$ and occurs only in the adversarial case where all nonzeroes are read from or redistributed to a single process. Algorithm 9 summarises the parallel ingestion process for matrices; that of vectors follows an analogue procedure. When choosing the shared-memory backend, the for-loops starting at lines 2 and 14 will be thread-level parallelised using a static schedule while local buffers are split into thread-level ones; for brevity we do not detail this.

Algorithm 8 Synchronise P local vectors into a global one.

Input: s, P, the local process ID and the total number of user processes
$x_s.local$, a process-local part of the input vector
n_s, the length of $x_s.local$
Output: $x_s.global$, the global output vector that concatenates all P local vectors

Buffer requirements: $\left(\lceil \frac{w_{\text{boolean}}}{w_{\text{indices}}} n \rceil + \max_k n_k\right) w_{\text{value}} + 4P w_{\text{size\_t}}$ bytes

1: Set nz_s to $x_s.local.nz$
2: Get $max\_nz = \max_k nz_k$, $min\_nz = \min_k nz_k$, and $x_s.global.nz = \sum_{i=0}^{P-1} nz_k$ {three allreduces}
3: Get $offset_s = \sum_{k=0}^{s-1} n_k$ {prefix-sum}
4: **if** min_nz equals n **then**
5: **for** $k = 0$ to $P - 1$ **do**
6: Get n_s elements from $x_k.local.values$ into $x_s.global.values$ at $offset_k$ {single allgather}
7: Set $x_s.global.assigned$ to *true* everywhere
8: Set $x_s.global.nz = n$
9: **else if** $w_{\text{boolean}} n < w_{\text{indices}} max\_nz$ **then**
10: **for** $k = 0$ to $P - 1$ **do**
11: Get n_s elements from $x_k.local.values$ into $x_s.global.values$ at $offset_k$
12: Get n_s elements from $x_k.local.assigned$ into $x_s.global.assigned$ at $offset_k$ {two allgathers}
13: Clear $x_s.global.stack$, set $x_s.global.nz$ to 0
14: **for** $i = 0$ to n **do**
15: **if** $x_s.global.assigned_i$ **then**
16: Add i to $x_s.global.stack$, increment $x_s.global.nz$
17: **else**
18: Set nz_s to $x_s.global.nz$
19: Get $offset_s = \sum_{k=0}^{P-1} nz_s$, get nz_k $\forall k$ {prefix-sum and allgather, overwrites old $offset_s$}
20: Clear $x_s.global$, retrieve buffers $inbuf_s$ and $outbuf_s$ of size nz and nz_s values, respectively
21: **for** $i = 0$ to $x_s.local.nz_s - 1$ **do**
22: Set $outbuf_{s,i}$ to $x_s.local.values_{x_s.local.stack_i}$
23: **for** $k = 0$ to $P - 1$ **do**
24: Get nz_k elements from $outbuf_k$ into $inbuf_s$ at $offset_k$
25: Get nz_k elements from $x_k.local.stack$ into $x_k.global.stack$ at $offset_k$ {two allgathers}
26: Set $x_s.global.nz$ to $\sum_{k=0}^{P-1} nz_k$
27: **for** $i = 0$ to $\sum_{k=0}^{P-1} nz_k - 1$ **do**
28: Set $index$ to $x_s.global.stack_i$
29: Set $x_s.global.assigned_{index}$ to *true*
30: Set $x_s.global.values_{index}$ to $inbuf_{s,i}$
31: **return** $x_s.global$

Data extraction in *SEQUENTIAL* mode must occur collectively and requires user processes to retrieve all remote data elements. This will hence incur $\Theta(nz)$ inter-process data movement for a parallel overhead of $\Theta(nzP)$, and should be avoided in practice. In parallel mode, iterators will only return locally stored data proportional to nz/P elements without any inter-process communication.

Error handling. As with the shared-memory parallel backend, most errors are handled unambiguously before any local work is executed. If one user process encounters the *ILLEGAL*, *MISMATCH*, or *UNSUPPORTED* error, for example, all other processes are guaranteed to encounter the same error while the offending call has no other effects beyond returning the error code. This occurs without inter-process coordination. By contrast, the *OUT_OF_MEMORY*, *IO_ERROR*, and *PANIC* error codes as well as exceptions thrown by container constructors or auxiliary iterators could originate from just one specific user process and must be handled locally. If mitigation proves impossible, a user can clean up and exit the local process gracefully. LPF ensures a premature local exit will lazily propagate as a *PANIC* error code at sibling processes, at the latest whenever such processes would normally exchange data with a failed process.

Algorithm 9 Parallel data ingestion into a distributed matrix.

Input: *start*, set of process-local forward iterators in start position
end, matching set of iterators in end position
mode, the requested IO mode
m, n, the global matrix size
s, P, the local process ID and the total number of processes
m_s, the local number of matrix rows to store (c.f. π)
b, a configurable input buffer size
Output: $A.local$, a pre-allocated local output matrix of size $m_s \times n$

Buffer requirements: $(2w_{\text{indices}} + w_{\text{values}})(b + recv\_nz) + 4Pw_{\text{size\_t}}$

Set nz_k^s to 0 for all $0 \leq k < P$, and set $offset_s$ to 0
for each nonzero $a_{ij} \in (start, end)$ **do**
  Increment $local\_nz$ by one
  if *mode* equals *PARALLEL* **or** $\pi(i) = s$ **then**
    Increment $nz^s_{\pi(i)}$ by one
if *mode* is *PARALLEL* **then**
  Get $recv\_nz = \sum_{k=0}^{P-1} nz_s^k$ {single prefix-sum}
else
  Set $recv\_nz = nz_s^s$
Retrieve local $buffer_s$ of size $recv\_nz$ nonzeroes and $2recv\_nz$ indices
if *mode* is *PARALLEL* **then**
  for $k = 0$ to $P - 1$ **do**
    Retrieve local $stack_k^s$ with max. capacity of b nonzeroes and $2b$ indices
for each nonzero $a_{ij} \in (start, end)$ **do**
  if *mode* is *SEQUENTIAL* **and** $\pi(i) = s$ **then**
    Push (i, j, a_{ij}) unto $buffer_s$
  else
    Push (i, j, a_{ij}) unto $stack^s_{\pi(i)}$
  if *mode* is *PARALLEL* **then**
    Set $full_s$ to *true* whenever $\exists k$ for which $|buffer_k^s| = b$, and *false* otherwise
    Get $full = \vee_{k=0}^{P-1} full_k$ {single all-reduce}
    if *full* **or** a_{ij} is the last nonzero in $(start, end)$ **then**
      Get $remote_k = |stack_s^k|$
      Get $local_k = \sum_{t=0}^{k-1} nz_s^t$ {allgather and prefix sum}
      for $k = 0$ to $P - 1$ **do**
        Get $remote_k$ triplets from $stack_s^k$ into $buffer_s$ at $offset_s + local_k$ {single alltoallv}
      Increment $offset_s$ by $local_P$, clear $stack_k^s$
grb::buildMatrix($A.local$, $buffer_s$, $buffer_s + recv\_nz$); {Delegate to single-process backend}
return $A.local$

4. Evaluation

To evaluate the C++ GraphBLAS API we express two canonical graph algorithms in Section 4.1. These algorithms can be executed with auto-vectorisation on a single core, on shared-memory parallel machines, distributed-memory parallel machines, or on hybrid shared- and distributed-memory machines by switching the backend the code compiles against; the algorithms as expressed remain unmodified. We additionally demonstrate the use of the performance semantics by systematically establishing a costing for both algorithms.

Section 4.2 then proceeds with experiments to show that the described API and implementations work correctly and efficiently, and do so at different scales. Weak and strong scaling experiments indicate that our performance semantics describe performance characteristics observed in practice, while throughput estimates on microbenchmarks and real-world datasets confirm our implementations reach close to peak performance.

4.1. **Algorithms.** We discuss the implementation of two algorithms based on our API: the PageRank (PR) in Algorithm 10 and the k-nearest neighbours (k-NN) in Algorithm 11. For brevity, we omit error

checking alike that on Alg. 10, line 9; every update of *rc* requires users to check it. For both algorithms we avoid having to instantiate temporary containers by having users provide the necessary ones. This allows for the reuse of temporary buffers. We perform input dimension checks (omitted) and clear user-supplied buffers prior to use. All algorithms return standard error codes. Algorithm output, input, and temporary container(s) are passed by reference.

Algorithm 10 The PageRank algorithm in GraphBLAS

Input: L, Any square grb::Matrix< void > of size n, the link matrix
α, IOType scalar, such that $(1-\alpha)$ is the random jump probability
tol, Any value of type IOType, the tolerance for deciding convergence
max, Any integer, the maximum number of iterations
x, y, r, grb::Vector< IOType > of size n, used during computation

Output: *pr*, a grb::Vector< IOType > of size n, the PageRank vector of L. May contain an initial guess on input.

grb::RC pagerank($pr, L, \alpha, tol, max, x, y, r$):

```
 1: grb::Semiring< IOType, IOType, IOType, IOType,
      grb::operators::add, grb::operators::mul,
      grb::identities::zero, grb::identities::one > ring;
 2: auto addM  = ring.getAdditiveMonoid();
 3: auto mulOp  = ring.getMultiplicativeMonoid().getOperator();
 4: grb::RC rc  = grb::SUCCESS;
 5: IOType δ, residual;
 6: size_t n = grb::nrows(L);
 7: if  grb::nnz( pr ) ≠ n  then
 8:    rc = grb::set(pr, 1/static_cast<IOType> (n));
 9: if rc equals grb::SUCCESS then
10:    rc = grb::set(x, 1);
11: rc = grb::clear(r);
12: rc = grb::mxv(r, L, x, ring);
13: rc = grb::eWiseLambda([&r, &α]( const size_t i ){
      if r[i] > 0
        r[i] = α/r[i]
      }, r);
14: for i = 0 to max do
15:    δ = residual = 0;
16:    rc = grb::foldl< grb::descriptors::invert_mask > ( δ, pr, r, addM );
17:    rc = grb::apply( x, pr, r, mulOp );
18:    δ = (α * δ + 1 − α)/static_cast< IOType > (n);
19:    rc = grb::set(y, 0);
20:    rc = grb::vxm< grb::descriptors::in_place > (y, x, L, ring);
21:    rc = grb::foldl( y, δ, addM );
22:    rc = grb::dot(residual, pr, y,
            grb::operators::add< double > (),
            grb::operators::abs_diff< double > ());
23:    if residual ≤ tol then
24:       break;
25:    std::swap(pr, y);
26: return rc;
```

The PageRank Algorithm 10 computes for an input graph L a score for every vertex that indicates its relative importance compared to other vertices. The algorithm modifies L into a stochastic matrix G, the Google matrix, and executes a power method to find the dominant Eigenvector which assigns each vertex its PageRank score. The modification to a stochastic matrix requires the introduction of a parameter $0 < \alpha < 1$, where $(1-\alpha)$ is most commonly interpreted as the probability users jump from one webpage to another regardless of whether there was a hyperlink from the former to the latter. Our listing corresponds to the canonical PageRank algorithm by Brin and Page [PBMW99, LM11].

Algorithm 11 k-Nearest Neighbours in GraphBLAS

Input: G, Any square grb::Matrix< T > of size n, the directed graph
$source$, An integer in $\{0, 1, \ldots, n-1\}$, the source vertex
k, Any integer, the k in k-NN
in, A grb::Vector< bool > of size n
Output: out, A grb::Vector< bool > of size n, the k-neighbourhood of $source$

grb::RC knn($out, G, source, k, in$):

```
1: grb::Semiring< bool, bool, bool, bool,
       grb::operators::logical_or, grb::operators::logical_and,
       grb::identities::logical_false, grb::identities::logical_true
   > booleanRing;
2: grb::RC rc = grb::clear(in);
3: rc = grb::clear(out);
4: rc = grb::set(out, true, source);
5: for i = 0 to k − 1 do
6:    std::swap(in, out);
7:    rc = grb::vxm< grb::descriptors::add_identity | grb::descriptors::in_place > (
          out, in, G, booleanRing
      );
8: return rc;
```

To initialise we require the out-degrees r of each vertex. We compute this by multiplying L with a vector of ones on lines 9–12. Since the type of the input matrix is void, the identity of the multiplicative monoid of the semiring substitutes for matrix nonzero values. For those vertices with nonzero out-degree, we pre-compute the propagation factor α divided by the out-degree on line 13. Line 16 computes the dangling factor which corrects the PageRank computation for zero out-degree vertices; it adds up the PageRank scores corresponding to dangling vertices by using r as an inverted mask. The subsequent line computes for all non-dangling nodes their current PageRank value multiplied with their propagation factor and stores the result in x, which is the input vector to the SpMV multiplication that follows. We account for the random jump probability and redistribute the PageRank contributions of the dangling nodes uniformly on line 21, and compute the updated residual in the one-norm on the subsequent line.

We employ the absolute-difference operator ($f(x, y) = |x - y|$) for the one-norm computation via a dot product. This operator is not associative nor does it have an identity, thus requiring the non-semiring version of the grb::dot. This primitive requires that its operands be dense, which the algorithm indeed forces on all of x, y, and pr. We omit all instances grb::descriptor::dense from the listing for brevity, but add this descriptor where-ever appropriate in our implementation: its use speeds up the algorithm's execution by eliminating run-time checks for sparsity.

Using the performance semantics defined in Table 6, Table 9 analyses the body of the main for-loop which calls six primitives: foldl, apply, set, vxm, foldl, and dot. Table 10 shows the same systematic costing in case of multiple user processes using the presented 1D backend. These costs add to the intra-node costs of Table 9 with n and nz in the latter table replaced by n/P and nz/P, respectively.

The definition of performance semantics is the first stage in the automatic derivation of costing tables such as constructed manually here. Such tables could become standard feedback to users who could use it to more easily assess whether the implemented algorithm may be improved and gauge what the performance characteristics in terms of problem sizes will be. This possibly also is a first step in allowing for automatic transformations to optimise GraphBLAS algorithms.

For the k-NN algorithm we wish to find, for an input graph G and a specified source vertex i, which other vertices are within k hops from a given $source$; i.e., find the k-nearest neighbours of the source vertex. We consider a vertex to be its own neighbour. To compute this we define a Boolean semiring and start out with a vector x_0 of size n containing only a single element $(x_0)_{source} = true$. To find the neighbour list of that source, we multiply x_0 with $G + I_n$ under the Boolean semiring, where I_n is the identity matrix of equal size to G. This ensures a vertex is indeed considered its own neighbour even if the diagonal of G were empty. This immediately yields the 1-hop neighbourhood $x_1 = x_0(G + I_n)$. Repeating this multiplication yields the k-hop neighbourhood as $x_k = x_0(G + I_n)^k$. We use $\mathcal{O}(1)$-cost std::swap on the input and output vectors. Of additional note is the use of the add_identity descriptor that reinterprets G as $G + I_n$, where the multiplicative identity of the given semiring defines the values of the diagonal I_n.

Cost	foldl	apply	set	vxm	foldl	dot	total
work	n	n	n	$nz+n$	n	n	$6n+nz$
operations	n	n	0	$2nz$	n	$2n$	$5n+2nz$
data movement							
-worst case	n	$3n$	$2n$	$4nz+n$	n	$2n$	$10n+4nz$
-best case	n	$3n$	$2n$	$2nz+3n$	n	$2n$	$12n+2nz$

TABLE 9. Systematic intra-process costing of the PageRank in Algorithm 10 using Tables 6 and 7, taking into account all vectors are dense. Different operator applications may differ in cost: the grb::dot, for example, has the add and absdiff operations that amount to one, respectively, two floating point operations (flops) each. Data movement is proportional to the size of the data type (IOType in Alg. 10) and the configured index data type.

Cost	foldl	vxm	dot	total
work	$P-1$	$n+P-1$	$P-1$	$n+3(P-1)$
operations	$P-1$	n	$P-1$	$n+2(P-1)$
data mov.	$P-1$	$P-1+n$	$P-1$	$n+3(P-1)$
synch.	1	1	1	3

TABLE 10. Systematic inter-process costing of the PageRank in Algorithm 10 for the 1D backend. Work and operator applications of order n/P are parallelised over T process-local threads. These costs are in addition to the intra-node costs for $n/P \times n$ matrices in Table 9.

The costing of the k-NN starts off with the two grb::clear calls at the start of the algorithm: these take $2n$ work, 0 operator applications, and $2n$ data movement. The single-element grb::set costs $\Theta(1)$ work and data movement. Let J_k be the set of nonzero coordinates of $x_k = x_0(G+I_n)^k$. By application the rules in Table 7 the ith grb::vxm costs $|J_i| + nz(A_{:,J_i})$ in work and data movement and $2nz(A_{:,J_i})$ in operator applications. In case of a threaded intra-process backend the work and operator application costs are divided by T after which all costs are increased with T. The distributed-memory backend adds $\sum_{i=1}^{k} nz(x_i)$ to operator application costs, $k(P-1) + \sum_{i=1}^{k} nz(x_i)$ to work and inter-process data movement, and k synchronisation costs.

Direction-optimisation (push/pull) [BAP12] may be implemented by adding a mask to Alg. 11, line 7:

$$rc = \text{grb::vxm} < \text{grb::descriptors::add\_identity} \mid \text{grb::descriptors::in\_place} \mid \text{grb::descriptors::invert\_mask} > (out, in, in, G, booleanRing).$$

Use or computation of this mask, however, is not free and may not pay off. Section 4.2.4 discusses push/pull variants, their costing, and resulting trade-offs.

4.2. **Experiments.** We perform experiments on the two machines summarised in Table 11. The processors have Intel Turbo disabled and hyperthreads enabled. All software threads are pinned to hardware threads during experiments. Both machines run Linux at kernel version 3.10 on the Ivy machine and version 4.18 on Cascade. All code is compiled using the GNU GCC compiler, version 4.8.5 on the Ivy machine and version 8.3.1 on the Cascade one. Distributed-memory experiments are run on a cluster of Ivy machines, connected via a single-switch Infiniband EDR network which has a theoretical throughput of 100 Gb/s. Inter-node communication is handled by the Lightweight Parallel Foundations (LPF) [SY19], which interfaces directly to the ibverbs driver for remote direct memory access. Our GraphBLAS implementation was configured with a SIMD width of 32 and 64 bytes on Ivy and Cascade, respectively. Indices referring to matrix nonzeroes are set to *size_t* while indices referring to vector elements are *unsigned int*, thus 8 and 4 bytes, respectively, on both our systems.

To establish a baseline for expected performance, Table 11 additionally reports the bandwidth achieved by 1) a vector-to-vector memory copy assuming the input vector has no temporal reuse, 2) a 'stream' kernel performing element-wise operations on up to three input vectors into one output vector, and 3) a 'reduce' kernel that performs similar element-wise operations but folds the result into a scalar. For the latter two experiments we apply different microkernels and report only the fastest results. The reported figures are an average over thirty runs while monitoring unbiased sample standard deviations to remain less than

	Ivy	Cascade	
2× Intel Xeon	E5-2690v2	6238T	CPUs
Core / CPU	10	22	
Memory / CPU	128	96	GB
Mem. channels / CPU	4	6	
Mem. speed / CPU	1600	2933	MT/s
L1 size / core	32	32	kB
L2 size / core	256	1024	kB
L3 size / CPU	25	30.25	MB
Speed / core	3	1.9	GHz
Hyperthread / core	2	2	
AVX unit / core	1 (AVX)	2 (AVX-512)	
Peak compute	480	2 675	Gflop/s
Peak throughput	190.7	262.2	Gbyte/s
Sequential nt-memcpy	11.0	8.83	Gbyte/s
Sequential stream	18.4	15.9	Gbyte/s
Sequential reduce	9.95	5.04	Gbyte/s
Single-socket nt-memcpy	35.1	77.1	Gbyte/s
Single-socket stream	37.9	96.2	Gbyte/s
Single-socket reduce	41.7	99.8	Gbyte/s
Double-socket nt-memcpy	76.7	153	Gbyte/s
Double-socket stream	75.6	194	Gbyte/s
Double-socket reduce	83.3	202	Gbyte/s

TABLE 11. Experiment machines specifications and baseline performances.

one percent of the measured average. In the subsequent text we refer to this percentage as the *relative standard deviation*. Vectors are large enough to ensure out-of-cache behaviour and the benchmark code is available freely[4].

Subsequent experiments with GraphBLAS follow the same methodology to avoid machine variability issues; we report both the average performance and its relative standard deviation over r_1 runs. The benchmark sleeps for one second between runs. To cope with small workloads of tens of milliseconds or less and to obtain a stable average time of code that employs dynamic OpenMP schedules, each run furthermore repeats the computation r_0 times. We hence average over $r = r_0 r_1$ repetitions, and by default set r_0 so that a single run takes at least 100 milliseconds while $r_1 = 30$. Unless otherwise noted, all experiments achieve a relative standard deviation of 2% or less by choosing larger r when required. To retain a meaningful sample standard deviation, we ensure that $r_1 \geq 10$ throughout all experiments.

To perform weak scaling experiments where problem sizes increase while compute resources remain constant, we select the Delaunay matrices [HSS10] from the SuiteSparse matrix collection [DH11]. These are 14 matrices $n = 10, 11, \ldots, 23$ of increasing scale that represent neighbour information resulting from Delaunay triangulation on random points in a unit square. We also experiment with large graphs from different real-world application areas summarised in Table 12, which sorts the datasets in increasing number of vertices. Vectors with a low number of vertices vectors tend to fit in cache, thus exhibiting higher performance relative to graphs with higher vertex counts. Due to their large size, MOLIERE_2016 (dataset 7) and Clueweb12 are too costly to be subject to our benchmarking methodology; for these we perform only two runs and discard the slowest, without computing a sample standard deviation. While this avoids some accidental outliers, this adapted methodology does not account for any performance variation due to dynamic scheduling.

Our experiments revolve around the two algorithms introduced in Section 4.1: the k-nearest neighbours (k-NN) and the PageRank. The former takes $k = 4$ and uses a fixed source vertex within an experiment, while the PageRank is seeded using a normalised unit vector and runs until convergence of the residual in the one-norm to 10^{-8} or less. We experiment with each algorithm using the three backends from Section 3 and present their results in Sections 4.2.1–4.2.3. Section 4.2.4 discusses trade-offs for (masked) sparse matrix–sparse vector multiplication, evaluates the heuristic proposed in Section 3.1 (page 13), compares it to alternatives, and details interactions between direction optimisation and our GraphBLAS traits system.

[4]See `http://www.multicorebsp.com`.

Dataset	ID	#vertices	#edges
Delaunay_nn	-	2^n	$\approx 6 \cdot 2^n$
coPapersCiteseer	1	434 102	32 073 440
wikipedia-20070206	2	3 566 907	45 030 389
channel-500x100x100-b050	3	4 802 000	85 362 744
adaptive	4	6 815 744	27 248 640
rgg_n_2_24_s0	5	16 777 216	265 114 400
road_usa	6	23 947 347	57 708 624
MOLIERE_16	7	30 239 687	6 669 254 694
europe_osm	8	50 912 018	108 109 320
com-Friendster	9	65 608 366	3 612 134 270
Clueweb12-graph	-	978 408 098	42 574 107 469

TABLE 12. Graphs used in experiments. All are taken from the SuiteSparse matrix collection, except for Clueweb12 [BV04]. Properties of the fourteen Delaunay matrices are parametrised in n with the number of vertices exact and the number of edges approximated. The Clueweb12-graph is exclusively used for distributed-memory parallel experiments in Section 4.2.3.

	Dot product	Reduction	Fused multiply-add
Hand-coded	120 (12.4)	106 (7.03)	199 (11.2)
GraphBLAS	120 (12.4)	106 (7.03)	204 (11.0)
eWiseLambda	125 (12.0)	131 (5.69)	205 (10.9)

TABLE 13. Dense sequential microbenchmarks on Ivy in milliseconds (and Gbyte/s). The row labelled 'GraphBLAS' indicates grb::dot, grb::foldl, and a grb::muladd in the cases of a dot product, reduction, and a fused element-wise multiply-add, respectively. Arrays are of size 100 000 000 (100M) doubles. All relative standard deviations are less than 1%.

	Dot product	Reduction	Fused multiply-add
Hand-coded	227 (6.56)	221 (3.37)	216 (10.3)
GraphBLAS	226 (6.59)	220 (3.39)	217 (10.3)
eWiseLambda	228 (6.54)	226 (3.30)	217 (10.3)

TABLE 14. Same as Table 13 but for Cascade. Figures are in milliseconds (and Gbyte/s). Performance generally is lower compared to Ivy since there is less bandwidth per core available. All dot product and reduction experiments reported have relative standard deviations less than 1%.

4.2.1. *Vectorising backend.* Within this subsection we experiment using only a single core of each machine, leaving all other cores idle. In Section 3 we claim operations are automatically vectorised whenever possible. We first validate this by comparing common dense computational patterns expressed in GraphBLAS versus hand-coded and compiler-optimised for-loops in Tables 13 and 14. The operations are a dot product ($\alpha = \sum_{i=0}^{n-1} x_i y_i$, grb::dot), a reduction ($\alpha = \sum_{i=0}^{n-1} x_i$, grb::foldl), and a fused multiply-add ($z_i = \alpha x_i + z_i$ for all $0 \leq i < n$, grb::muladd); with α a double-precision floating-point scalar (a *double*) and with x, y vectors of doubles of length n. The eWiseLambda, for which the GraphBLAS implementation cannot aid the compiler in vectorising the computational work, is benchmarked as well. As expected, hand-coded loops perform equally well to GraphBLAS equivalents while the grb::eWiseLambda performs slightly worse. These results furthermore are in-line with the baseline maximum performance of the stream baseline in Table 11, while the reduce baseline indeed acts as an upper bound on reduce-like operations such as grb::foldl. The grb::dot corresponds to an in-place stream operation with scalar output, and indeed achieves a performance in-between the reduce and stream baselines on both architectures.

Next, we look at a weak scaling experiment using the Delaunay datasets where both the number of vertices and edges successively double. Given the costings of the 4-NN and PageRank, run-times should

		Ivy		Cascade	
n	b (×)	k-NN (×)	PR (×)	k-NN (×)	PR (×)
10	182	0.049	0.0139	0.0293	0.0166
11	298 (1.64)	0.054 (1.10)	0.0285 (2.05)	0.0374 (1.28)	0.0448 (2.70)
12	373 (1.25)	0.0605 (1.12)	0.0734 (2.58)	0.0459 (1.01)	0.116 (2.59)
13	398 (1.07)	0.0669 (1.11)	0.160 (2.18)	0.0464 (0.828)	0.259 (2.23)
14	307 (0.78)	0.0668 (1.00)	0.330 (2.06)	0.0384 (0.809)	0.539 (2.08)
15	405 (1.32)	0.0884 (1.32)	0.668 (2.02)	0.0450 (1.17)	1.08 (2.00)
16	160 (0.40)	0.0999 (1.13)	1.35 (2.02)	0.0331 (0.736)	2.16 (2.00)
17	204 (1.28)	0.187 (1.87)	2.70 (2.00)	0.0537 (1.61)	4.30 (1.99)
18	191 (0.94)	0.404 (2.16)	5.40 (2.00)	0.112 (2.09)	8.60 (2.00)
19	266 (1.39)	1.75 (4.33)	11.1 (2.06)	0.150 (1.34)	17.3 (2.01)
20	107 (0.40)	1.99 (1.14)	23.7 (2.14)	0.332 (2.21)	35.1 (2.03)
21	331 (3.09)	5.87 (2.95)	53.5 (2.26)	0.666 (2.01)	70.7 (2.01)
22	117 (0.35)	9.92 (1.69)	116 (2.17)	0.915 (1.37)	146 (2.07)
23	118 (1.01)	12.8 (1.29)	235 (2.03)	2.15 (2.35)	298 (2.04)

TABLE 15. Weak scaling of the vectorising backend on the Delaunay_nn datasets. We report the neighbourhood size b and the run-time in milliseconds for the 4-NN and PageRank algorithms. These are each followed by their increase versus the previous dataset in parentheses. The run-time for the PageRank is per iteration.

successively double as n increases. For the PageRank this is unconditionally true while for the k-NN this is only true if $b/n \to 0$ as n increases, where b is the size of the 4-hop neighbourhood. For the Delaunay graphs b varies from 160 (for $n = 2^{16}$) to 405 ($n = 2^{15}$). For the PageRank algorithm we expect caching effects for smaller n: multiple vectors of 2^{10} doubles fit in L1 data cache on both Ivy and Cascade, while for larger n the vectors cannot be cached. The k-NN algorithm requires less storage per vector and may exhibit higher performance for lower n: two vectors with *void* data types fit L1 cache for $n < 12$, L2 for $n < 15$ (Ivy) and $n < 17$ (Cascade), L3 for $n < 22$ (Ivy), and L2+L3 for $n < 23$ (Cascade). The input matrix size exceeds L3 cache size for $n > 19$ and for $n > 20$ on the Ivy and Cascade machines, respectively; parts of the input matrix may hence be reused from cache across various runs for smaller n.

Table 15 reports the results. For the 4-NN we observe that run-times generally increase with b and n. For both architectures and when most data can be cached ($n < 16$) increases are far less from the expected 2×. Anomalies occur for $n = 19$ and $n = 20$: their corresponding run-times increased disproportionately with b and n. These respective n coincidence with matrices no longer being cacheable in L3 on both architectures. More modest increases to run-times are visible for $L3$ at $n = 22$ and $n = 23$ on Ivy, resp., Cascade due to vectors not being cacheable. Apart from these effects, the increase in run-time for $n > 15$ tends to 2× and is lower only whenever b decreased as well. Since our vectorising backend switches between scatter and gather automatically and not all edges of the input graph need be traversed, effective performance (in operator application per second) and effective throughput (in byte/s) cannot be estimated from the reported timings.

The PageRank results conform to expectation except for the smallest n where associated vectors fit in cache, thus speeding up those computations beyond our prediction. Since the number of flops is proportional to the number of nonzeroes of the data set and all vectors are dense, we can derive the effective performance as well as infer bounds on the effective throughput. Maximum performance for Ivy is 1.32 Gflop/s for $n = 10$, corresponding to an effective bandwidth between 12.0 and 17.9 Gbyte/s, assuming all or none of the vector elements are cached, respectively. The largest dataset attains the lowest performance at 0.643 Gflop/s and 5.86–8.71 Gbyte/s. For Cascade the maximum throughput is 1.11 Gflop/s for $n = 10$, corresponding to 10.1–15.0 Gbyte/s. The lowest measured throughput is 0.507 Gflop/s for $n = 23$, corresponding to 4.62–6.88 Gbyte/s. Since for $n = 10$ vectors fit in cache the lower bounds of 12.0 and 10.1 Gbyte/s apply, which exceed both the sequential nt-memcpy and reduce baselines of Table 11 on both architectures, indicating the vectorising backend achieves high performance. For $n = 23$ vectors do not fit cache, hence effective throughput should approach the upper bounds of 8.71 and 6.88 Gbyte/s. This is close to the sequential reduce baseline for Ivy and exceeds it on Cascade, again indicating good performance.

We proceed with the real-world datasets, listed with IDs 1 to 9 in Table 12. On Ivy, vectors corresponding to the data set with ID 1 fit into L3 cache exactly while those corresponding to dataset 2 and

	Ivy			Cascade		
Dataset	k-NN	PageRank		k-NN	PageRank	
ID	ms.	ms.	Gbyte/s	ms.	ms.	Gbyte/s
1	4.52	45.9	6.23– 16.5	4.08	68.3	4.19– 11.1
2	78.8	318	2.27– 4.21	74.1	270	2.67– 4.96
3	10.3	131	8.84– 18.0	3.90	141	8.20– 16.7
4	13.4	174	5.39– 7.13	3.72	125	7.51– 9.94
5	24.2	575	6.58– 13.0	7.59	848	4.47– 8.83
6	33.7	615	4.90– 5.72	15.0	792	3.81– 4.44
7	180 000	61 600	0.860–2.47	153 000	56 100	0.943–2.71
8	64.6	1 150	5.46– 6.20	27.5	1 150	5.46– 6.20
9	103 000	51 600	0.658–1.68	86 500	34 900	0.973– 2.48

TABLE 16. Experiments on real graphs from different application areas on Ivy and Cascade, for the k-NN and PageRank (PR) algorithms using the vectorising backend. Timings are in milliseconds and per iteration for the PR algorithm. For PR, we add throughput bounds in Gbyte/s. For the k-NN results, $k = 4$.

3 fit within a factor two of L3 cache size. For Cascade with its exclusive cache architecture, all datasets with ID lower than 5 have corresponding dense vectors that fit in cache. All other datasets would require a more than 2× larger combined cache size to fit. We expect a significantly higher performance when vectors can be cached, and, conform to Table 11, expect higher performance for Ivy compared to Cascade. The results in Table 16 confirm these intuitions.

For the k-NN Cascade outperforms Ivy except of the largest dataset, with ID 7, which seems contrary to the baselines in Table 11. This dataset has the highest number of nonzeroes per row amongst all datasets, and hence is less sensitive to memory access latencies that we conclude dominate the k-NN costs on these architectures. For the PageRank, once caching of vectors is not a factor (IDs 5 and higher), Ivy performs faster than Cascade as expected, except for datasets 7 and 9. This indicates a relatively larger number of cache misses hampering performance, supported also by the significantly lower bandwidth attained on those two datasets on both architectures. Large sparse matrices with irregular structures such as indeed knowledge bases (datasets 2 & 7) and social networks (9), are known to achieve much lower performance peaks due to cache misses [YB09]. For the other matrices we achieve bandwidth ranges in line with the sequential nt-memcpy baseline for datasets where vectors cannot be cached, while well exceeding it when caching is possible.

By comparing the k-NN results versus the PageRank ones we can furthermore gauge the effectiveness of the sparse matrix–sparse vector multiplication. For k-NN we distinguish two cases: datasets where the neighbourhood after $k = 4$ hops remains small relative to the total number of vertices n, and those with a neighbourhood size close to n. Datasets 7 and 9 fit the latter category as the relative neighbourhood size is larger than 90 percent, while for datasets 3–6 and 8 these are less than 1 percent. Datasets 1 and 2 achieve close to 8 and 26 percent, respectively. For small neighbourhoods the time taken to compute four hops is well below the time required for a single PageRank iteration; Algorithm 5 hence exploits the sparsity of the input vector well. For the high neighbourhood cases our implementation switches to Algorithm 4, which should ensure the time taken for the 4-NN is bounded by 4× that of a PageRank iteration. Indeed, a 4-NN search takes about 3× the time taken for a PageRank iteration for datasets 7 and 8 on Cascade, while these are 3× and 2× for those datasets, respectively, on the Ivy machine.

4.2.2. *Shared-memory parallel backend.* We consider OpenMP parallelisation over both one and two sockets with and without using hyperthreads, and distinguish between configurations by qxr, where q and r are the number of sockets and hyperthreads per socket used, respectively, for a total of $T = qr$ threads. As with Section 4.2.1, Table 17 starts with microbenchmarks on the dot product, reduction, and fused multiply-add primitives and compare versus hand-written code parallelised using OpenMP. These microbenchmarks use all available cores since no significant parallel overheads in T occur, save for a final reduction for the dot and reduce benchmarks. We hence expect throughputs close to the dual-socket baselines in Table 11. Unlike for the vectorising backend and due to the reduction phase in the dot and reduce benchmarks, the eWiseLambda can only be applied for the fused multiply-add (FMA). This achieves 22.1 and 34.8 GB/s on Ivy and Cascade, respectively, both using hyperthreads for the fastest result. The slowdown versus the GraphBLAS and hand-coded variants are significant and highlight the importance of vectorisation.

	Ivy			Cascade		
	Dot	Reduction	FMA	Dot	Reduction	FMA
hand	37.0 (40.3)	10.7 (69.6)	81.9 (27.3)	22.8 (65.4)	6.34 (118)	29.6 (75.5)
grb	20.0 (74.5)	10.6 (70.3)	41.8 (53.5)	12.3 (121)	6.19 (120)	24.4 (91.6)

TABLE 17. As Tables 13 and 14, but using OpenMP for multi-threading for the compiled codes and using the shared-memory parallel backend for the GraphBLAS primitives. The reported numbers correspond to the use of all cores with hyperthreads, except for the hand-written dot product on both machines where the reported results without hyperthreads perform better.

Only for the reduction do the hand-written codes result in kernels with comparable performance to that of GraphBLAS-generated code. The only difference is that our GraphBLAS implementation uses template meta-programming to generate more easily vectorisable code, a step that results in close to a factor-two performance gains for the FMA on Ivy and the dot product on Cascade. FMA performance is not close to stream baseline since the former performs an out-of-place computation, whereas the latter employs in-place computations that achieve higher performance. The dot reaches 91.7 and 81.6 percent of the nt-memcpy baseline performances on the Ivy and Cascade machines, respectively, while the reduce reaches 97.1 and 82.3 percent.

We briefly discuss weak scaling experiments on the Delaunay datasets in text only. Since the neighbourhood sizes are limited to 405, the work required for the 4-NN is not enough to enable any speedups for $n < 19$. For larger n, the linear part of the k-NN costing becomes dominant enough to offset the low amount of work in the grb::vxm calls: we then observe speedups using a single socket without hyperthreads, and do so on the Ivy machine only. The highest speedup is observed for $n = 21$ at 1.78× versus the sequential timing reported in Table 15. Indeed, the neighbourhood size for $n = 21$ is relatively large at 331, while, for example, at $n = 23$ it is 118, resulting in a slowdown of 0.96×. Though untested, the optimal number of threads likely lies below that of the number of cores on a single socket; this also clarifies why no speedup at all is observed on the Cascade machine: it has a significantly higher core count compared to the Ivy machine.

Since the PageRank consists of more work, parallelisation using one socket shows benefit from $n > 10$ and $n > 11$ onward on the Ivy and Cascade machines, respectively, and shows benefit using two sockets from $n > 12$ and $n > 14$ onward. Table 18 shows speedups across both architectures and using multiple OpenMP parallelisations. As noted in prior work, straightforward OpenMP-based parallelisation does not scale well on NUMA systems [YR13]; indeed the shared-memory parallel backend does not exploit multiple sockets well, even for $n = 23$. The best results are achieved using one socket without hyperthreading for $n < 18$ and with hyperthreading for $n > 18$. The lowest measured throughput range on one socket without hyperthreads is 6.43–9.55 Gbyte/s on Ivy and 1.92–2.85 Gbyte/s on Cascade for $n = 10$. The highest measured throughput range is 37.7–56.0 Gbyte/s (single socket, without hyperthreads) and 41.2–61.3 Gbyte/s (single socket, with hyperthreads) on Ivy and Cascade, respectively; these bounds match the single-socket nt-memcpy baselines from Table 11. Figure 1 confirms the PageRank costing behaviour as the increase of the time per iteration for successive n approaches 2× in all configurations.

Also on the real-world datasets parallelisation of the k-NN requires a large enough neighbourhood size. We hence distinguish between datasets that have a neighbourhood of 1 percent or more of the total number of vertices (IDs 1, 2, 7, and 9) and those with less than 1 percent (IDs 3–6 and 8). The high-neighbourhood datasets have speedups reported in Table 19. The highest speedup attained on the low-neighbourhood datasets is 1.25× on dataset 3 using one socket without hyperthreads for the Ivy machine. No speedups were observed on the Cascade machine for these datasets since its higher core count would require more work for efficient parallelisation.

Finally, we present results of the PageRank experiments on the real-world datasets in Table 20. On Ivy, the best performance is found using both sockets with hyperthreads for 40 threads total, with the exception of datasets 7 and 8. On Cascade, in contrast, the best performance is found at 44 threads allocated to the same single socket, also using hyperthreads– there, single exceptions occur for datasets 2 and 7 where two sockets without hyperthreads are faster. The highest throughputs are obtained on dataset 1, attaining a lower bound on throughput of 58.7 and 78.4 Gbyte/s on the Ivy and Cascade machines, respectively. On datasets with vectors that cannot be cached, the highest measured throughputs are in the range of 30.3–61.9 Gbyte/s (dataset 3, dual socket) and 43.8–86.6 Gbyte/s (dataset 5, single socket) on Ivy and Cascade. Since cache effects here should be limited, we approach the upper bounds of these ranges–

	Ivy				Cascade			
n	1x10	1x20	2x10	2x20	1x22	1x44	2x22	2x44
10	0.534	0.168	0.144	0.0727	0.190	0.0983	0.0642	0.0637
11	0.965	0.312	0.281	0.148	0.466	0.259	0.144	0.109
12	**1.91**	0.753	0.651	0.353	0.937	0.632	0.431	0.203
13	**2.84**	1.39	1.18	0.710	**2.10**	1.31	0.556	0.436
14	**3.76**	2.18	1.77	1.24	**3.70**	2.32	0.849	1.41
15	**4.18**	3.04	2.30	2.00	**5.15**	3.63	2.33	2.28
16	**4.48**	3.54	2.63	2.56	**6.21**	5.08	3.42	3.33
17	**4.72**	3.97	2.81	2.81	**7.20**	6.68	4.36	2.96
18	**4.73**	4.38	2.97	2.81	7.84	**7.94**	5.07	3.49
19	3.89	**4.35**	3.09	3.33	7.79	**8.08**	5.99	4.99
20	3.41	**3.98**	3.04	3.37	7.37	**8.21**	6.11	5.08
21	3.26	**3.88**	3.10	3.57	7.43	**8.47**	6.71	6.52
22	3.21	**3.66**	3.15	3.43	6.90	**7.70**	6.84	6.77
23	3.21	**3.36**	3.14	3.51	6.56	**7.27**	6.77	6.88

TABLE 18. The speedup of the PageRank computation on the Delaunay matrices using the shared-memory parallel backend using the results from the sequential vectorising backend in Table 15 as a base line. We employ one and two sockets with and without hyperthreads on both architectures. All relative standard deviations are at 2 percent or less except for 1x20 for $n = \{11, 12\}$ on the Ivy machine, which stand at around 3 percent and could not be reduced further. Highest speedups on each machine are displayed in bold.

Dataset	Relative	Ivy				Cascade			
ID	neighb. size	1x10	1x20	2x10	2x20	1x22	1x44	2x22	2x44
1	8.25%	**1.07**	0.96	0.57	0.37	0.831	**1.04**	0.287	0.187
2	26.4%	0.86	**1.98**	1.69	1.55	2.47	**2.98**	1.36	0.946
9	92.0%	2.23	2.33	2.12	**8.73**	16.5	**18.9**	14.6	15.4
7	96.9%	6.50	9.52	7.66	**9.89**	17.3	18.6	19.6	**23.1**

TABLE 19. Speedups of the 4-NN algorithm versus their sequential baseline in Table 16 for those datasets with a relative neighbourhood size of 1% or more. Rows are ordered according to increasing neighbourhood size, with highest speedups displayed in bold.

Dataset	Ivy				Cascade			
ID	1x10	1x20	2x10	2x20	1x22	1x44	2x22	2x44
1	7.33	7.24	7.76	**9.43**	14.3	**19.1**	17.8	16.0
2	6.42	7.58	6.70	**8.50**	10.5	10.8	**10.9**	10.5
3	3.16	2.97	3.08	**3.43**	6.08	**6.64**	5.81	5.35
4	2.88	2.98	2.75	**3.31**	3.77	**4.04**	3.66	3.56
5	3.79	3.78	3.74	**4.27**	8.61	**9.84**	9.39	9.14
6	3.16	3.13	3.06	**3.66**	6.62	**7.47**	7.01	7.22
7	5.41	**6.84**	5.67	6.55	11.5	10.5	**22.6**	22.3
8	2.94	**3.39**	2.83	3.30	4.75	**5.36**	4.98	5.02
9	8.65	12.4	9.72	**14.8**	16.0	**16.7**	11.7	11.7

TABLE 20. Speedups of the PageRank algorithm versus their sequential baseline in Table 16.

which are close to or exceed the respective nt-memcpy baseline performances. Even though the use of two sockets theoretically doubles the peak throughput we observe such 2× gains only for the PageRank on dataset 7 on the Cascade architecture.

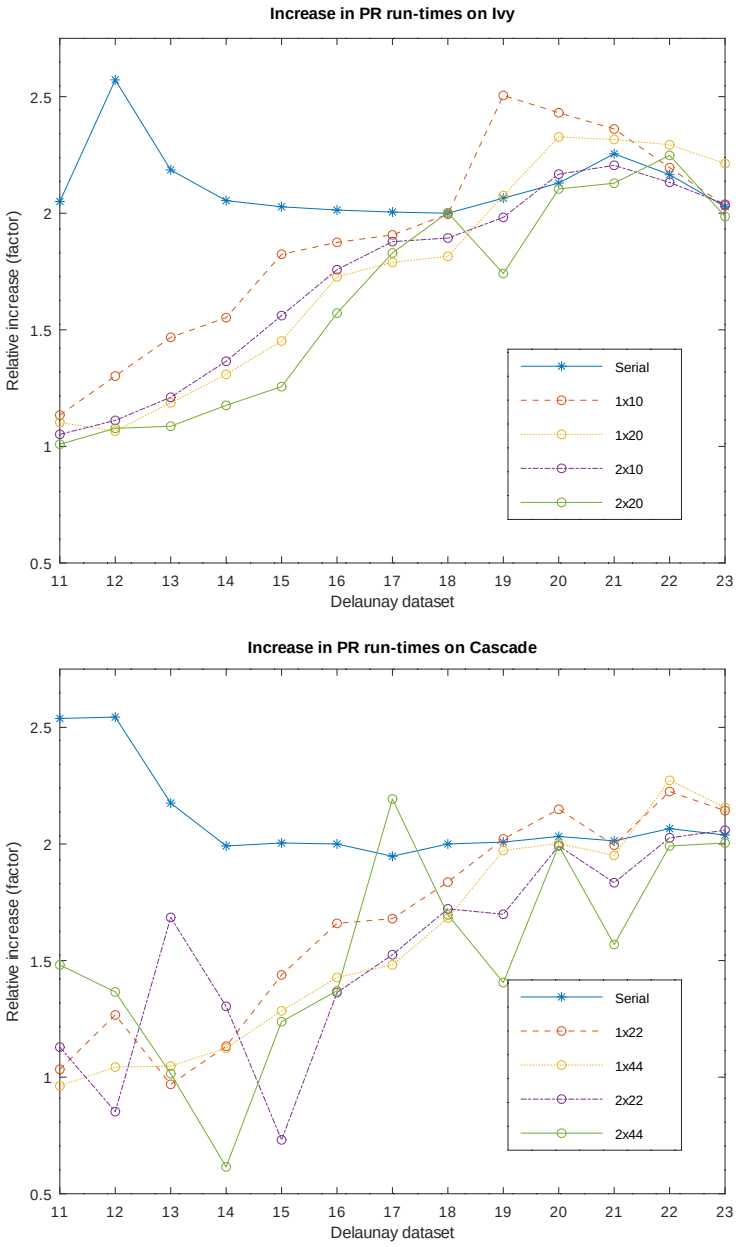

FIGURE 1. The increase in time per iteration for $n > 10$ relative to the preceding n for the PageRank (PR) computation, using the shared-memory parallel backend in various configurations on the Ivy (top) and Cascade (bottom) machines.

4.2.3. *Distributed-memory parallel backend.* We couple the distributed-memory parallel backend as described in Section 3.3 with the shared-memory parallel backed described in Section 3.2. This yields a fully hybrid distributed- and shared-memory parallel GraphBLAS backend that uses LPF and OpenMP. The main use cases for a distributed-memory backend are to 1) process datasets that do not fit memory of a single machine, and 2) to process data faster. Neither the scale of the Delaunay datasets nor those of datasets 1–9 mandate the use of multiple nodes for either reason. This section hence focuses on the link structure of the Clueweb12 dataset, resulting from a large-scale crawl of the world-wide web executed by the Lemur project [Lp12]. Its link structure was derived via the Webgraph project [BV04], resulting in one of the largest publicly available irregular datasets available, containing 42 billion edges. We require at least three Ivy Bridge nodes to be able to process this dataset in memory, and perform a strong scaling

	Ivy nodes			
	3	4	5	6
Input	2340	1490	1140	**977**
k-NN	115	**39.1**	46.6	60.8
PR	24.5	**21.8**	23.7	24.1

TABLE 21. Strong scaling experiment of the k-NN and PageRank (PR) algorithms on three to six Ivy Bridge nodes using the Clueweb12 dataset. Timings are in seconds and per iteration for the PR. For the k-NN, $k = 4$.

experiment up to six nodes to investigate its speedups. These nodes are connected via Infiniband EDR. LPF orchestrates communication directly through ibverbs.

Table 21 summarises our strong scaling results. We report the time for parallel input of the Clueweb matrix. This is the only experiment where use the parallel I/O features introduced with our API. These timings include all operations in Algorithm 9, thus including both process-local I/O as well as interspersed communication. The Clueweb12 dataset is read from MatrixMarket format and has a disk size of 786 GB, thus resulting in effective ingestion speeds of 344–805 MB/s for 3–6 nodes. We observe the largest decrease in ingestion time between 3 and 4 nodes at 1.57×, and the lowest decrease from 5 to 6 nodes at 1.17×. The efficiency of parallel I/O is indeed expected to monotonically decrease as the all-to-all communication phase starts to dominate the cost of the distributed reads; the latter scales linearly with the number of nodes P, while the former incurs a cost of $(P-1)nz/P = \Theta(nz)$ and does not scale with P. The computational loads hit their best performance at $P = 4$ for both algorithms; also here, parallel overheads start to overtake parallelised work as P increases. The k-NN, in contrast, may yield fortunate work distributions for varying P. The 4-hop neighbourhood sizes for Clueweb12 lie at less than 1% of the total number of vertices, which means that the sparsity of the input vector drives the computational load during each iteration. Without analysing the input matrix nonzero structure, such effects cannot be quantified beforehand and occur unpredictably at run-time. We can, however, measure these effects after the fact using the relative parallel efficiency. The efficiency of the 4-NN for $P = 4$ relative to $P = 3$, for example, is $\frac{115 \cdot 3}{4 \cdot 39.1} = 2.21$. Since this is significantly higher than the normally ideal 1, we indeed conclude that for $P = 4$ we observe a much improved effective work distribution.

4.2.4. *Automatic push/pull.* In Section 4.1 we briefly showed how we may implement direction optimisation [BAP12] in GraphBLAS by the use of the invert_mask descriptor and the use of the input vector x_k as a mask for computing $x_{k+1} = x_k(G + I_n)$, as also noted by Yang et al. [YBO18]. By using algebraic type traits, the non-masked vxm in Algorithm 11 can be automatically transformed into the masked variant by recognising that 1) the requested operation is in_place, and 2) the addition operator is a logical-or. These two conditions imply nonzeroes in the output vector are immutable, and thus that the SpMV multiplication need not consider nonzero output elements.

However, the inversion of a mask is not free. Our reference implementation does not maintain a list of where zeroes are present, which means all n elements must be inspected individually. If we had a list of locations of zero positions, nonzero positions could still store explicit zeroes. Inversion would thus still require $\Theta(n)$ work, unless also supplied with the structural_mask descriptor and a user guarantee that A contains no explicit zeroes. To ensure a sublinear cost where possible we previously experimented with the unmasked k-NN (Alg. 11), noting that this still performs direction optimisation: a very sparse input vector will employ CCS-based SpMV multiplication, making use of the scattering inner kernel (Alg. 5). This is the 'push' direction. If, however, the number of nonzeroes in the input vector times T is larger than n, we perform the multiplication using CRS and the gathering kernel (Alg. 4), which corresponds to the 'pull' direction. A third natural variant considering that CRS is a standard data structure in sparse matrix computations would use CRS-based multiplication (pull) always.

We compare the 4-NN performance for these three variants and experiment using the three largest real-world datasets. These datasets have a relative neighbourhood size of 96.9 (dataset 7), 92.0 (9), and less than 1 percent (8). For the former two push/pull strategies should prove effective while for the latter only push should perform best. We include dataset 2 with a middle-ground relative neighbourhood size of 26.4 percent. Table 22 shows that the explicit computation of a mask for the full push/pull variant is not always worth the cost and that our default implementation indeed benefits of direction optimisation without explicit mask computation, thus providing a good trade-off.

	Sequential			OpenMP		
Dataset ID	pull only	push/pull full	push/pull ours	pull only	push/pull full	push/pull ours
7	418	187	**180**	18	**12.2**	18.2
9	243	193	**103**	13.3	17.5	**11.8**
2	1.09	1.13	**0.0788**	0.0816	0.0853	**0.0397**
8	5.00	5.69	**0.0646**	0.380	0.429	**0.0829**

TABLE 22. Comparison of k-NN approaches on the Ivy machine for the vectorising backend (left) and the shared-memory parallel backend (right). Timings are in seconds with the fastest highlighted bold. The OpenMP results correspond to the configuration from 1x10, 1x20, 2x10, and 2x20 that achieves the lowest timing. Results from our push/pull correspond to those of Tables 16 and 19.

5. Related Work

We separate related work between specification and implementation, starting with the former.

5.1. **Specification.** The interface and general concepts of the C++ GraphBLAS interface described here are close to that of the GraphBLAS C API [BMM+17], changed mostly to suit standard C++ and generic programming principles. We briefly list major differences and novelties. Container element types are template parameters instead of a type identifier constant. This makes new type definitions trivial: any plain-old-data and non-GraphBLAS container type is a valid container element type. Similarly, users may define new operators by declaring a templated class implementing the custom operator and detailing which traits it adheres to (such as associativity or commutativity) and use a standard-defined wrapper class to turn it into a GraphBLAS compatible operator with functioning type traits. This ensures the operator gets compiled into high performance codes, as opposed to being applied as-is at run-time via a function pointer. I/O via STL iterators successfully separate concerns regarding high-performance parsers, differing file formats, buffering, and so on from core GraphBLAS functionality. We allow for parallel I/O through iterators. While the Combinatorial BLAS (CombBLAS) [BG11] also allowed for parallel I/O, they did not do so through iterators. We introduced user processes to the GraphBLAS specification not only for I/O, but also to specify data-centric operations in an SPMD setting, specify parallel error handling, and specify inter-process overheads of algebraic operations executed on distributed-memory.

We define foldl and foldr primitives instead of a single reduce primitive since the 'direction' of applying the operator matters if the domains $T1$, $T2$ of an operator $f : T1 \times T2 \to T3$ differ. We allow for non-monoid reductions for dense vectors. Additionally, we allow for non-associative binary operators though disallow, through type traits and static asserts, their use in primitives where associativity is required. We removed the difference between blocking and non-blocking execution modes because a non-blocking implementation can be provided as a backend separate from a blocking one. Such a backend furthermore would not require wait-like primitives, and in non-blocking execution allow any optimisation that retains the defined performance semantics as an upper bound on the cost incurred at run-time. Still, it may frustrate algorithm designers to relinquish such a level of control over performance. We therefore specify the *grb::eWiseLambda* with transparent performance semantics, allowing users to define explicitly what should be fused, as well as when and how it should be fused. The C API includes functions presently unsupported by our work, such as Kronecker products and the ingestion of data with duplicate inputs.

From a programming language point of view, we note parallels between the C++ concepts of *policies* and the way we implemented descriptors as template arguments to GraphBLAS primitives. Policies are small classes that describe narrow behaviours or structures, that, when combined, describe complex aggregate behaviours [Ale01]. Indeed, the descriptors currently supported convey how masks should be interpreted, how input matrices should be interpreted, et cetera; these various orthogonal behaviours and reinterpretations are combined into a single descriptor, modifying the overall behaviour of GraphBLAS primitives they are passed to. We note that exposing explicitly additional mathematical concepts beyond those described here will support application areas beyond graph computing. Kepner and Hayden identify a set of concepts that underpin the broader area of Big Data [KJ18], while the use of algebraic concepts in general programming has been investigated thoroughly by Stepanov [SM09, SR14] and has strong commonalities with the GraphBLAS approach, especially when done in C++. Programming using algebraic structures such as semirings goes back at least to Aho, Hopcroft, and Ullman in 1974 [AHU74].

5.2. **Implementation.** The GraphBLAS C API [MKH18] was first implemented by IBM, which released a C API wrapper around their Graph Processing Interface in 2018 [KHK+18]. Davis followed with the open SuiteSparse:GraphBLAS in 2019 [Dav19] and its recent OpenMP parallelisation [ACD+20]. On the C++ side, McMillan et al. proposed a C++ interface and implementation targeting GPUs (GBTL) [ZZL+16], while Yang et al. created a GraphBLAS C++ API around the GPU-based Gunrock graph processing framework (GraphBLAST) [YBO19]. Extremely relevant remains the Combinatorial BLAS by Buluç et al. [BG11], which pioneered many of the design elements now in use.

We discuss key differences between our vectorising and shared-memory parallel backends and the above-mentioned work. The SuiteSparse implementation stores matrices by means of a single CRS, which we employ Gustafson's format. While this incurs a 2× storage overhead versus CRS, it does allow the flexibility to heuristically decide which multiplication variant to select. Though the trade-offs between these variants are clear from a matrix computing perspective (see Sections 3.1 and 4.2.4), the same concept seen from a purely graph computing perspective proved more intricate and led to direction-optimisation and push/pull techniques [BAP12, BPG+17] and graph processing frameworks such as Ligra, which, like our work, automates the choice [SB13]. Both Davis and GraphBLAST opt to store the nonzero structure of vectors in a sorted fashion, whereas we have opted for an array and stack to efficiently support the most common operations without having to sort. This goes back to sparse accumulator techniques by Gilbert et al. [GMS92]. The parallelisation strategies of the SuiteSparse implementation using OpenMP are similar to ours except that whenever load-imbalance may occur, our implementation uses dynamic schedules whereas Aznaveh et al. exploit sorted nonzero structures via a binary search to derive static load-balanced work partitions. Both the sorting and binary search incur overheads that our implementation does not. Those overheads are unaccounted for, or, rather, disallowed by the performance semantics we attached to each GraphBLAS primitive.

Our work is similar to GBTL, GraphBLAST, and CombBLAS in the common choice of a C++ GraphBLAS interface. GBTL and our work make the similar choice of using generic programming techniques, while GraphBLAST remains closer to the C API specification and CombBLAS employs object-oriented principles. In our work, and alike GBTL, domain types are part of the semiring. This differs from the other C++ interfaces. Instead of passing descriptors as an argument (e.g., GraphBLAST), our interface passes it as a template argument to algebraic primitives. McMillan et al. instead opt for views that wrap around containers [BBM+20a], which is compatible with the way our code processes descriptors at compile-time and thus similar. Views are arguably also closer to standard C++ paradigms since they strongly relate to type traits we introduced in our work.

6. Conclusions

We present a C++ GraphBLAS specification in-line with generic programming standards that exposes algebraic type traits and adds performance semantics. User program in a sequential, data-centric fashion using linear algebraic operations over scalars, vectors, and matrices parametrised by operators, monoids, and semirings. The GraphBLAS system then takes care of high performance computational (HPC) details by auto-vectorising and auto-parallelising the given linear algebraic expressions, enabling users to benefit from HPC performance while not requiring any of the associated programming and algorithmic expertise. Our described implementations indeed achieve close to peak performance on several of the detailed benchmarks, and do so while conforming to performance semantics attached to each GraphBLAS operation. We added new algebraic type traits that allow for compile-time transformations for higher efficiency and improve user feedback on algebraic errors.

We define strict performance semantics for single multi-threaded user processes, derived from asymptotic bounds of state-of-the-art algorithms and data structures. Our implementations prove that the defined semantics are well achievable in practice, and, as experiments show, without detriment to performance. The eWiseLambda provides tighter control of data movement when required. For multiple user processes inter-process costings must be implementation defined: this allows different backends to optimise for different use cases. Using these performance semantics, users can systematically attach costings to algorithms they express. While our cost semantics are not exact, they nevertheless enable 1) evaluation of whether a given algorithm is optimal or whether it may yet be improved for efficiency, and 2) prediction of performance characteristics in varying inputs and backends.

Implementations may contain multiple backends, and must provide at least one. Different backends can add support different hardware or optimise for different use cases. We present three backends: a vectorising, a shared-memory parallel, and a distributed-memory parallel one. Any GraphBLAS algorithm can execute using a backend selected at compile time or at run time. We implement two canonical

graph algorithms: the k-nearest neighbourhood and the PageRank. These employ the sparse matrix–dense vector and the sparse matrix–sparse vector kernels, respectively, and as such incur very different performance characteristics. Using these two algorithms and their systematic costings we confirm our implementation behaves in line with the specified performance semantics through weak scaling experiments for the vectorising and shared-memory parallel backend and through strong scaling experiments for the distributed-memory parallel backend. Our implementation leads to quantifiable good performance on two different x86 architectures, close to peak performance for vector–vector operations as well as for the PageRank algorithm on some of the real-world datasets experimented with.

7. Future work

This present work notably excludes a discussion of a generalised sparse matrix–sparse matrix multiplication operation, a grb::mxm. The CombBLAS has shown its applicability for various graph algorithms and shows a scalable implementation of such algorithm requires a 2D distribution [BG11]. This is also the case for algorithms based on sparse matrix–vector (SpMV) multiplications, even on shared memory with an increasing number of NUMA domains [YBRM14]. Two-dimensional Cartesian partitionings such as employed by the CombBLAS, however, do not suffice for speeding up SpMV multiplication on highly NUMA systems. Previous work [YR13, YBRM14] instead relies on sparse matrix partitioners such as Mondriaan [BAY$^+$12], PaToH [ÇA11], or Zoltan [DBH$^+$02] that are able to exploit latent sparsity structures and improve significantly on the theoretically optimal bounds for random sparse and dense matrices [BAY$^+$12, KB20].

An orthogonal argument in favour of integrating general sparse matrix partitioning methods with the GraphBLAS and with the sparse matrix–sparse matrix multiplication in particular, comes from sparse and graph neural network inference. Casting this in linear algebraic terms amounts to a sequence of mxm operations [KKM$^+$17, DAK19]. Recent work by Pawłowski et al. found that efficient inference requires tiling through the multiple mxm operations, using a joint partitioning of all weight matrices involved to block for maximum cache reuse [PBUY20]. The use of Cartesian distributions, though sufficient for the scalable computation of a large class of mxm-based GraphBLAS algorithms such as the triangle counting algorithm by Azad et al. [ABG15] cannot enable this high performance inference mechanism. Since the cost of data movement over a network increases beyond that of movement within a shared-memory machine [BAY$^+$12], we furthermore believe that upcoming distributed-memory GraphBLAS efforts [BBM$^+$20b] and new distributed-memory backends stand to benefit most of integration with general 2D partitioning methods– such integration, however, is not straightforward and may require modification of the GraphBLAS interface.

The sequential and shared-memory backends can be improved by incorporating blocked sparse matrix storages such as Compressed Sparse Blocks (CSB) [BFF$^+$09] or compressed incremental storages [YR13] in favour of CRS+CCS. CSB, when assuming hypersparsity of blocks, achieves a storage requirement of $\Theta(nz + m)$. That bound is equal to CRS, while enabling the use of the cache-oblivious orders within blocks. Compressed incremental storages under the same assumption improve on the standard CSB and CRS storage bounds. They also enable the use of cache-oblivious curves between sparse blocks, instead of within. Both storages support both ELLPACK and segmented-reduce type vectorisation when required by the underlying hardware [Yze15]. Such enhancements must, however, take into account the use of permutation matrices: recent work by Spampinato et al. has shown that their addition to the GraphBLAS enables the efficient implementation of depth-first search [SSL19]. This enlarges the space of algorithms that can be efficiently implemented using the GraphBLAS, and low-level optimisations such as those on the data structure level should not preclude this.

While a distributed-memory backend supporting general 2D partitionings would be ideal, a 2D block cyclic distributed-memory backend would enable scalable mxm-based algorithms without having to incur a sparse matrix partitioning phase. Such backend may reuse Algorithms 7 and 8 to apply only on the rows, resp., columns of a 2D $Q \times R$ process mesh, instead of having these algorithm apply to all $P = QR$ processes as described for our 1D block-cyclic backend. Finally, the three presented backends are optimised for speed; equally important would be a backend that optimises memory footprints instead. Our systematic costing methodology enabled by our specified performance semantics allows for the automatic generation of analytic performance profiles of user programs. This, in turn, leads to the opportunity of automatic backend selection, matching algorithms and their inputs to a suitable backend at run time.

References

[ABG15] Ariful Azad, Aydın Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 804–811, New York, 2015. IEEE, IEEE.

[ACD+20] Mohsen Aznaveh, Jinhao Chen, Timothy A Davis, Bálint Hegyi, Scott P Kolodziej, Timothy G Mattson, and Gábor Szárnyas. Parallel GraphBLAS with OpenMP. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 138–148, Philadelphia, 2020. SIAM, SIAM.

[AHU74] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Boston, 1974.

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, Boston, 2001.

[BAP12] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, New York, 2012. IEEE, IEEE.

[BAY+12] Rob H Bisseling, B O Fagginger Auer, A N Yzelman, Tristan van Leeuwen, and Ümit V Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In *Combinatorial Scientific Computing*, pages 321–349. Chapman & Hall/CRC Press, London, 2012.

[BBM+20a] Benjamin Brock, Aydın Buluç, Timothy G Mattson, Scott McMillan, and José E Moreira. A roadmap for the GraphBLAS C++ API. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 219–222, New York, 2020. IEEE, IEEE.

[BBM+20b] Benjamin Brock, Aydın Buluç, Timothy G Mattson, Scott McMillan, José E Moreira, Roger Pearce, Oguz Selvitopi, and Trevor Steil. Considerations for a distributed GraphBLAS API. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 215–218, New York, 2020. IEEE, IEEE.

[BFF+09] Aydın Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, New York, 2009. ACM.

[BG11] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[BMM+17] Aydın Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. The graphblas c api specification, v1.0, 2017.

[BPG+17] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104, New York, 2017. ACM.

[BV04] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[ÇA11] Ümit V Çatalyürek and Cevdet Aykanat. PaToH (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, New York, 2011.

[CHPvdG07] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[DAK19] Timothy A Davis, Mohsen Aznaveh, and Scott Kolodziej. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via SuiteSparse:GraphBLAS. In *2019 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, New York, 2019. IEEE, IEEE.

[Dav19] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.

[DBH+02] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–97, 2002.

[DH11] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[GMS92] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[HSS10] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, New York, 2010. IEEE.

[KB20] Timon E Knigge and Rob H Bisseling. An improved exact algorithm and an NP-completeness proof for sparse matrix bipartitioning. *Parallel Computing*, 96:102640, 2020.

[KBB+15] 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.

[KG11] J Kepner and J Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, Philadelphia, 2011.

[KHK+18] M. Kumar, W. P. Horn, J. Kepner, J. E. Moreira, and P. Pattnaik. IBM POWER9 and cognitive computing. *IBM Journal of Research and Development*, 62(4/5):10:1–10:12, 2018.

[KHW+14] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.

[KJ18] Jeremy Kepner and Hayden Jananthan. *Mathematics of big data: Spreadsheets, databases, matrices, and graphs*. MIT Press, Boston, 2018.

[KKM+17] Jeremy Kepner, Manoj Kumar, José Moreira, Pratap Pattnaik, Mauricio Serrano, and Henry Tufo. Enabling massive deep neural networks with the GraphBLAS. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, New York, 2017. IEEE, IEEE.

[LBGR12] Adam Lugowski, Aydın Buluç, John R Gilbert, and Steve Reinhardt. Scalable complex graph analysis with the knowledge discovery toolbox. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5345–5348, New York, 2012. IEEE, IEEE.

[LM11] Amy N Langville and Carl D Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton university press, Princeton, 2011.

[Lp12] The Lemur project. Clueweb12 data set, 2012.

[LV15] Weifeng Liu and Brian Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350, New York, 2015. ACM.

[MKH18] José E Moreira, Manoj Kumar, and William P Horn. Implementing the GraphBLAS C API. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 298–309, New York, 2018. IEEE, IEEE.

[MS03] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114 – 152, 2003. 1998 European Symposium on Algorithms.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford, 1999.

[PBUY20] Filip Pawłowski, Rob H Bisseling, Bora Uçar, and A N Yzelman. Combinatorial tiling for sparse neural networks. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, New York, 2020. IEEE, IEEE. Accepted.

[SB13] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, New York, 2013. ACM.

[SC19] Conrad Sanderson and Ryan Curtin. Practical sparse matrices in C++ with hybrid storage and template-based expression optimisation. *Mathematical and Computational Applications*, 24(3):70, 2019.

[SM09] A A Stepanov and P McJones. *Elements of programming*. Addison-Wesley Professional, Boston, 2009.

[SR14] Alexander A Stepanov and Daniel E Rose. *From mathematics to generic programming*. Pearson Education, New York, 2014.

[SSL19] Daniele G Spampinato, Upasana Sridhar, and Tze Meng Low. Linear algebraic depth-first search. In *Proc. of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 93–104, New York, 2019. ACM.

[SY19] W. Suijlen and A. N. Yzelman. Lightweight parallel foundations: a model-compliant communication layer, 2019.

[YB09] A N Yzelman and Rob H Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.

[YBO18] Carl Yang, Aydın Buluç, and John D. Owens. Implementing push-pull efficiently in graphblas. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[YBO19] Carl Yang, Aydin Buluc, and John D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. In *ACS HPC and Data Analytics Workshop*, pages 1–39, New York, 8 2019. IEEE.

[YBRM14] A N Yzelman, Rob H Bisseling, Dirk Roose, and Karl Meerbergen. MulticoreBSP for C: a high-performance library for shared-memory parallel programming. *International Journal of Parallel Programming*, 42(4):619–642, 2014.

[YLZZ14] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: yet another SpMV framework on GPUs. *ACM SIGPLAN notices*, 49(8):107–118, 2014.

[YR13] 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.

[Yze15] A. N. Yzelman. Generalised vectorisation for sparse matrix–vector multiplication. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, New York, NY, USA, 2015. ACM.

[ZZL+16] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 912–920, New York, 2016. IEEE, IEEE.

Computing Systems Laboratory, Huawei Zürich Research Center, Switzerland
Email address: albertjan.yzelman@huawei.com

Visiting Researcher, Huawei Paris Research Center, France
Email address: danieldinardo@gmail.com
Email address: jonathan.mark.nash@gmail.com

Huawei Paris Research Center, France
Email address: wijnand.suijlen@huawei.com