Humble Heroes

A. N. Yzelman

Abstract

The concept of a *humble programmer*, coined by Dijkstra in the 1970s, fully acknowledges the complexity of programming, deeming it unsuitable for the human mind. This concept leads to the design of humble programming frameworks that not only simplify the implementation of algorithms, but also automate the optimization and scalability of algorithms across compute units. Such humble frameworks typically favor productivity over performance.

By contrast, *hero programmers* are experts with in-depth knowledge of parallel algorithms, applicable lower bounds, coding, and hardware, typically seeking to achieve the highest possible efficiency on any given system. Given that hardware complexity and diversity are set to increase in the years to come, *humble programmers* will not be able to drive increasingly more complex and larger-scale systems using contemporary software technologies. This, combined with the scarcity of hero programmers at the present time and in the foreseeable future, highlights the need for humble programming models that can achieve good performance.

This paper starts by outlining the successful humble MapReduce and Pregel vertex-centric frameworks and identifies the common design factors that they share with the novel *Algebraic Programming (ALP)* paradigm. Satisfying the need for supporting multiple humble programming models while reducing demands on hero programmers, this paper proposes a vertex-centric programming model on top of ALP, demonstrating that multiple humble programming models can be supported by a single software stack. Thus, fewer heroes may support a greater number of humble programmers.

Experiments demonstrate that the resulting ALP/Pregel paradigm scales well on a shared-memory parallel system, achieving speedups of up to 17.8x on common graph workloads. Even though vertex-centric algorithms that solve a given problem commonly differ from their canonical solutions, this paper compares such an algorithm (used for ranking web pages) with a highly optimized canonical solution for completing the same task. The result shows that in sequential execution ALP/Pregel achieves up to a 8.99x speedup, outperforming the canonical solution in 12 out of 13 datasets tested. In shared-memory parallel execution, ALP/Pregel achieves a 0.276–5.76 times speedup over a highly optimized implementation of the canonical PageRank algorithm. The slowdowns, which are down to 0.276x, are mostly due to the workspace not fitting in L3 cache for the vertex-centric algorithm. For the smallest and largest datasets where such cache effects are less apparent, ALP/Pregel is faster in 3 out of 6 instances with speedups of up to 5.73x. This indicates that humble algorithms may still achieve hero performance, contributing a solution to the looming crisis in software productivity.

1 Introduction

The perhaps esoteric title of "Humble Heroes" relates to one of the major challenges in computing: programming potentially highly parallel and highly heterogeneous resources, without complicating the task to the point where only a small percentage of highly skilled programmers can write efficient programs.

Writing software grows more and more difficult due to the increasing complexity of computer architectures, coupled with a diverse and growing range of architectures that programs should cope with. The recent hardware trends show increase both in the number of non-uniform memory access (NUMA) designs and the degree of non-uniformity — there are already four to eight NUMA domains on a modern CPU — while both the per-core memory capacity and bandwidth are decreasing. These factors will significantly increase pressure on future software design for any given architecture. Additionally, heterogeneity may appear on a wide spectrum from a single chip consisting of different types of processing units to large-scale data centers connecting a diverse range of architectures. Within and between these extremes, heterogeneity will realistically continue to appear.

With the diversification of architectures and the increase of their design complexity and heterogeneity, contemporary software technologies no longer scale. Maintaining standard libraries, for example, requires dedicated programmers with theoretical, algorithmic, domain, programming, and hardware expertise in order to achieve acceptable performance on a given architecture. The required breadth of expertise for maintaining standard libraries sets this class of programmers apart from the majority — colloquially, we call them *hero programmers*.

1.1 Humble Programming

In contrast to hero programmers, Dijkstra first coined the concept of a *humble programmer*. Paraphrasing his monologue [Dij72], this concept entails *approaching programming while fully recognizing the incredible complexity of the task, by preferring modest and elegant programming languages that respect the limitations of the human mind*. Perhaps implicit in humble programming, there is acceptance of a performance trade-off, favoring programming productivity over performance. This paper, however, provides evidence that performance and productivity need not be mutually exclusive.

Contemporary examples of humble programming models that scale across large-scale computational resources include MapReduce [DG08], Pregel [MAB+10], and Spark [ZCF+10], all of which make it possible to drive complex large-scale systems using simple, elegant, and easy-to-use abstractions that any programmer can understand. These programming models have been extremely successful in providing parallel compute power to humble programmers, allowing them to quickly deploy and scale up a variety of workloads.

1.2 The Software Bottleneck

Hero programmers, by contrast, strive to achieve peak performance — or as close to it as possible — for any given workload on any given architecture. They embrace the complexity that comes with the task, using low-level interfaces such as assembly language, intrinsics, threading interfaces, remote direct memory access, and message passing. The resulting code requires significant time and effort to build and is costly to maintain. Updating the code to novel versions of the architecture it was originally written for remains costly, while supporting completely novel architectures requires a ground-up duplication of all these efforts.

If the future is to be increasingly heterogeneous in nature, then the future software stack must not be dependent on hero programmers: there will never be enough expert programmers to support the many application domains that require mapping to different architectures. Instead, the future software stack must speak primarily to humble programmers, be deployable on different architectures, and achieve good performance — all while avoiding the software bottleneck.

1.3 Outline

Section 2 reviews the MapReduce and the vertex-centric Pregel [MAB+10] programming interfaces, providing a more in-depth characterization of the properties of successful humble programming models. Section 3 describes the novel *Algebraic Programming (ALP)* humble programming paradigm, and in particular ALP/GraphBLAS, which achieves hero-level performance on both shared- and distributed-memory systems [YDNNS20].

Section 4 notes that, while ALP may be a significant leap forward for anyone comfortable with using algebraic

annotations to express algorithms, some humble programmers may prefer to use alternative paradigms. The section asserts that neither forcing a single paradigm on humble programmers nor supporting an unchecked set of potentially many humble paradigms solves the challenge of productively using future large-scale heterogeneous systems. Instead, it advocates the alternative approach this paper takes: automatic translation of one humble programming model into another.

Section 5 realizes this ideal for Pregel and ALP, by introducing a C++ API for translating vertex-centric programming into ALP at compile time, thus supporting multiple humble paradigms with a single software stack. Section 6 presents experimental results, showing that the cost of simulating a Pregel interface on top of ALP is negligible and that such an interface achieves comparable performance to direct programming in ALP/GraphBLAS — it even outperforms direct programming if some relaxations are permissible. Section 7 concludes this paper and presents a future outlook.

2 Influential Humble Programming Models

This section discusses two pivotal humble programming models that have made massively parallel computing accessible to the public rather than to a limited group of parallel programming experts: MapReduce and Pregel.

2.1 MapReduce

In MapReduce, data items are key-value pairs (k, v) from domains K and V, respectively, and programs are a sequence of two alternating phases: *map* and *reduce*.

Let $D_0 \subset K \times V$ be the set of initial data items. The i-th *map* phase takes each key-value pair d_k from D_{2i}, where the integer k is in the range of $[0, |D_{2i}|)$, and maps it to a possibly empty set of new pairs $f_i(d_k)$. Then, one of the following may happen: the original pair d_k is filtered out, one new pair is generated, or an arbitrary number of new pairs is generated. The mapping function $f_i : K \times V \to \mathcal{P}(K \times V)$ is user-defined and may be different for each round i. The original set of data D_{2i} is discarded and replaced by the set of transformed entries D_{2i+1}.

The i-th *reduce* phase operates on all keys in the current data set D_{2i+1}, specifically, the set $K_i = \{k \in K \text{ s.t. } \exists (k, v) \in D_{2i+1}\}$. For a key k, assume there are $r > 0$ key-value pairs $(k, v_0), \ldots, (k, v_{r-1})$, then the reduction operation computes $v' = \odot_{i=0}^{r-1} v_i$ and produces an entry (k, v') in $D_{2(i+1)}$. The entry replaces all key-value pairs with key k in D_{2i+1} — this repeats for each key in K_i. As with the user-defined mapping function, the reduction operator $\odot$ is user-defined and may differ for each round.

While the mapping operations may be arbitrary, it is helpful if the reduction operators are associative, in which case the reduction can be automatically parallelized by systems like ALP; otherwise, the *reduce* phase can only be parallelized over each key $k \in K_i$, limiting the amount of parallelism exposed.

The round-based nature of a MapReduce communication, which consists of the *map* phase and the *reduce* phase, is closely related to the bulk synchronous parallel (BSP) paradigm of parallel computing [Val90]. The *map* phase induces no communication between parallel compute units whereas the *reduce* phase does require communication. BSP models a parallel system as local compute units with local memory, interconnected by a full-duplex network. Each local compute unit executes a program in two rounds — like MapReduce does. Each round includes two phases: 1) local computations using local memory elements, and 2) arbitrary data movement patterns between memory elements. On any one compute unit, a next round may only proceed when all incoming and outgoing messages have been received and sent, respectively. This does not imply that a global synchronization barrier exists between rounds.

There are two differences between direct-mode BSP and MapReduce: 1) BSP operates at the coarse granularity of one task per process while MapReduce operates at the granularity of the dataset sizes $|D_i|$; 2) BSP allows arbitrary communication patterns while MapReduce always sorts on keys. L. G. Valiant [Val90], however, proposed an *automatic mode* of BSP that allows the simulation of PRAM algorithms on a BSP architecture. The hashing mechanism employed in this mode is typically mirrored in MapReduce implementations such as Hadoop [SKRC10], in Pregel realizations such as Giraph [Ave11], and in the Spark framework [ZCF+10].

Such simulation techniques come at a cost and preclude the application of communication-optimal algorithms. Thus, orders of magnitude performance differences between humble programming frameworks and the direct-mode BSP are not uncommon. Suijlen and Yzelman made a performance comparison between ALP/GraphBLAS and Spark [SY19] (when both use ten nodes) and reported

a two-orders of magnitude performance difference for the PageRank algorithm on a moderate data size (of approximately 298 million edges).

2.2 Pregel

Pregel is a programming model for graph computations [MAB+10]. Assume the data graph $G = (V, E)$, then Pregel allows defining a round-based computation for each vertex $v \in V$ to execute. Each round consists of two steps: 1) execution of the user-defined program on each vertex; and 2) exchange of messages via the edges E. Messages are broadcast from vertices $v \in V$, that is, a message $m \in M$ is sent to all neighbors $N(v) = \{w \in V \mid (v, w) \in E\}$ of *v*.

Such *vertex-centric* programming models have achieved great success and inspired a significant number of vertex- and edge-centric programming frameworks [MWM15]. The round-based nature of vertex-centric programming, like that of MapReduce, is also a variation of the BSP model — like MapReduce, vertex-centric programming applies parallelism in a fine-grained fashion by mapping the programs on the input data itself. Most variants of Pregel follow the principle that while the algorithm is strictly round-based, the execution of such a program need not be. As such, latency may be hidden by overlapping the message communication phase of one set of vertices with the computational phase of other sets of vertices. This resonates with Valiant's automatic-mode BSP, which performs similar latency hiding through overlapping communication and computation phases [Val90].

Consider, as an example of a vertex-centric program, the problem of determining the strongly connected components (SCCs) of an undirected graph $G = (V, E)$, that is, identifying the minimum number of subsets $V_k \in V$ where (V_i, E), are strongly connected[1]. A simple vertex-centric program determines these subsets in two stages: 1) Each $v \in V$ is assigned a unique identifier; 2) During each round, each vertex sends its current identifier to its neighboring vertices, and overwrites the current identifier to the largest value contained in all incoming messages if that value is strictly larger than the current identifier. If the current identifier is not overwritten in the second stage, the vertex votes to halt the program. When all vertex programs vote to halt the program, the execution terminates. Once the execution terminates, the number of SCCs k corresponds to the number of unique identifiers that remain, and the component to which each vertex belongs is indicated by its identifier.

This vertex-centric algorithm and its termination criterion are intuitive to understand and therefore fit the bill of a humble solution for solving the SCC problem. However, if the input to this algorithm is an undirected line graph, the number of rounds required will be $|V|$, which means that $|V|$ vertex programs will be active per round, and that the work for this SCC algorithm amounts to $\Omega(|V|^2)$. By contrast, the Awerbuch-Shiloach [SV80, AS87] algorithm for SCC can achieve $\mathcal{O}(|E| \log |V|)$ work complexity. Algorithms with such improved bounds can be implemented on top of a vertex-centric framework but will be far less humble in nature [SW14]. The same holds for ALP and GraphBLAS; consider, for example, the linear-algebraic variant of Awerbuch-Shiloach by Zhang et al. [ZAB20].

Since adversarial graph structures such as line graphs do not naturally occur in the real world, the quadratic algorithm may be acceptable in practice — in which case the humble choice suffices. This paper, in line with its motivation, thus considers only easy-to-understand humble algorithms, leaving the important task of finding and realizing asymptotically optimal algorithms to the hero programmers.

2.3 Common Factors

The concepts shared by the successful MapReduce and Pregel programming models are as follows: First, they both operate on sets of data and express parallelism mainly by operating concurrently on data elements within those sets, which is known as a *data-centric* approach. Second, both their ways of expressing parallelism are implicit. Programmers express operations by mutating one set of data into another without considering their parallelization. Third, operations proceed round-by-round, thus exposing a sequential view of the final programs. This is similar to the direct-mode of BSP and traditional imperative programming. Fourth, efficient implementations of these two models exploit the imposed program structure to enable both scalable execution and a high degree of overlap between executing rounds and phases — this is achieved by overlapping communication with computation and exploiting parallel slackness [Val90, MAB+10]. These four concepts are furthermore also shared with another successful humble programming model: Spark [ZCF+10].

[1] A graph or subgraph is strongly connected only if there is a path between any two of its vertices.

3 Algebraic Programming

The ALP paradigm provides a data-centric, sequential, and imperative programming approach based on algebraic concepts and annotations. Designed to be humble, this paradigm presents three algebraic concepts to programmers: containers, structures, and primitives. ALP/GraphBLAS, introduced by Yzelman et al. as a C++ realization of the GraphBLAS [YDNNS20, BMM+17, BBM+19], focuses on sparse linear algebra. In ALP/GraphBLAS, containers take the form of vectors and matrices; structures can be binary operators, monoids, or semirings; and operations can be element-level applications of a binary operator, a matrix–matrix multiplication, and so on.

3.1 Algebraic Containers

Containers in ALP, when declared, are initially *empty* — meaning that they contain no values. By default, the minimum capacity of a container equals its maximum dimension. That is, a newly created vector of size n holds no values but has the capacity for n values. Similarly, a novel $m \times n$ matrix holds no values, and its default capacity is at least $\max\{m, n\}$. Larger or smaller capacities may be set when new containers are created, while existing containers can be resized to hold more or fewer values through `grb::resize`. An ALP implementation may assign capacities that are larger than requested, but not smaller. If a requested minimum capacity cannot be guaranteed, an error will be returned.

Using standard template library (STL) compatible iterators to C++ STL containers, data can be *ingested* into ALP containers via the two primitives: `grb::buildVectorUnique` and `grb::buildMatrixUnique`. The 'Unique' suffix in these primitives indicates that a source container shall not contain duplicate entries, i.e., multiple values shall not map the same container coordinate. The current size of a vector is returned via `grb::size`, the row size and column size of a matrix are returned via `grb::nrows` and `grb::ncols`, respectively, and the current number of values in a container can be referenced via `grb::nnz`. All values can be erased via `grb::clear`.

The following three statements are examples of creating a vector with the default capacity, a vector with potentially smaller capacity, and a matrix with the exact initial capacity as returned by a matrix file parser, respectively:

```
grb::Vector< double > x( n );
grb::Vector< bool > s( n, 1 );
grb::Matrix< void > A( m, n, parser.nz() );
```

Note that the element type of an ALP container appears as a template argument, as with the STL.

The extraction of data from ALP containers proceeds using iterators that can be retrieved through the `begin`, `cbegin`, `end`, and `cend` functions. These output iterators also conform to the STL, but support only the **const**–variants, that is, values in containers cannot be adapted through iterators. In addition to data ingestion through iterators, programmers may also set a vector to be a dense one with all its values set to a specific given value:

```
grb::set( x, 1.0 );
```

Likewise, one may set a single element of a container:

```
grb::setElement( s, true, n / 2 );
```

In the preceding examples, x corresponds to a dense vector $(1, 1, ..., 1)$, while s corresponds to a sparse vector where only one nonzero exists at position $n/2$ with the value (*true*).

3.2 Algebraic Structures and Algebraic Type Traits

An example of a binary operator is addition of double-precision floating point numbers, which ALP exposes as a C++ class template:

```
grb::operators::add< double >
```

An operator $\odot$ may have algebraic properties such as associativity $(a \odot (b \odot c) = (a \odot b) \odot c)$, commutativity $(a \odot b = b \odot a)$, and idempotency $(a \odot a = a)$. ALP/GraphBLAS exposes such properties through *algebraic type traits*, for example,

- `grb::is_associative< grb::operators::add< double > >::value, which reads true;`
- `grb::is_commutative< grb::operators::divide< float > >::value, which reads false;` or
- `grb::is_idempotent< grb:operators::min< unsigned int > >::value, which reads true.`

Algebraic type traits can be inspected at compile-time, thus enabling semantic checks and compile-time optimizations guided by algebraic properties.

Richer algebraic structures include monoids and semirings, which may be *composed* of operators and identities. For example,

```
grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >
    grb::identities::zero, grb::identities::one
>
```

describes the standard numerical addition and multiplication over doubles, also called the *plus-times* semiring. Operators each have a domain name attached as a template argument. Identities must have an element in those domains and otherwise the code will not compile.

For any binary operator $\odot : D_1 \times D_2 \to D_3$, the three domains potentially differ. Take a look at the following example:

```
typedef grb::operators::argmax<
    std::pair< size_t, float >,
    std::pair< size_t, float >,
    size_t
> ArgmaxUINT_FP32;
```

This example describes the argmax operator over tuples of integers and floating point numbers, resulting in an integer as per

$$argmax((i_0, \alpha_0), (i_1, \alpha_0)) = \begin{cases} i_0, \text{ if } \alpha_1 \leq \alpha_0 \\ i_1, \text{ otherwise.} \end{cases}$$

Such binary operators may still form monoids or semirings, for example,

```
grb::Monoid< ArgmaxUINT_FP32,
    grb::identities::infinity >
```

depicts a valid monoid where "infinity" over unsigned integers will be interpreted as the maximum representable value *maxint*, while "infinity" over a tuple is composed by recursion over the tuple types — which, in this case, resolves to $(maxint, \infty)$.

3.3 Algebraic Primitives

An algebraic primitive combines containers and structures, modifying the former in a way that depends on the latter. Perhaps the most simplistic operation takes two input vectors and generates one output vector by applying a given binary operator in an element-wise fashion. For example,

```
grb::Vector< double > y( n );
grb::eWiseApply( y, x, x,
    grb::operators::add< double >() );
```

computes $y = x \odot x$, where $\odot$ is given by the algebraic structure provided (simple numerical addition, in this example). Using the earlier examples that defined and populated x and after executing the above, y reads $(2, 2, \ldots 2)$.

Consider element-wise application in the sparse case:

```
grb::Vector< double > d( n, 1 );
grb::operators::assign_left_if< double, bool, double >
myOp;
grb::eWiseApply( d, x, s, myOp );
```

While x is dense, s contains only a single nonzero. Because the algebraic structure of a simple binary operator does not allow for the interpretation of missing values from a container, ALP will only apply the requested binary operation on nonzeroes x_i and s_i that appear on the same coordinate i, ignoring any values in x that do not have a matching value in s and vice versa. Consequently, the above results in a single entry, namely, $d_{n/2} = x_{n/2}$.

An associative operator, joined with an identity, forms a monoid, which allows algebraic primitives to interpret missing values in sparse containers. The following example shows the behaviors of numerical multiplication as an operator rather than a monoid, under element-wise application:

```
grb::Vector< double > oneTwo( n, 1 );
grb::setElement( oneTwo, 2.0, n/2 );
grb::operations::mul< double > mulOp;
grb::Monoid<
    grb::operations::mul< double >,
    grb::identities::one
> mulMon;
grb::eWiseApply( y, x, oneTwo, mulOp );
// y = oneTwo
grb::eWiseApply( y, x, oneTwo, mulMon );
// y = (1,..., 1,2,1,...1)
```

The `grb::eWiseApply` is an out-of-place primitive. The `grb::foldl` and `grb::foldr` provide in-place variants instead:

```
grb::foldl( y, oneTwo, mulMon );
// y = (1,..., 1,4,1,...,1 )
```

Some operations require richer algebraic structures. Consider, for example, the sparse matrix–vector (SpMV) multiplication $y = Ax$:

```
grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >,
    grb::identities::zero, grb::identities::one
> mySemiring;
grb::clear( y );
grb::mxv( y, A, x, mySemiring );
```

ALP defines that only `grb::set` and `grb::eWiseApply` can operate in out-of-place fashion, whereas all other primitives have in-place semantics. For example, the above `grb::mxv` sets y to $y \oplus Ax$ — with $\oplus$ being the additive operator of the given semiring — and does not set y to Ax. If the latter is intended, the output container must be cleared first, as in the above example.

All primitives with container output support masking. Since the vector s evaluates **true** only at position $n/2$,

```
grb::mxv( y, s, A, x, mySemiring );
```

requests only the computation of $y_{n/2}$, leaving any other elements as-is. We may also invert the effective mask through *descriptors*, which provide mechanisms that modify the interpretation of input containers such as masks, and will prove useful later in this paper. The following example updates all entries of y *except* the one at position $y_{n/2}$:

```
grb::mxv< grb::descriptors::invert_mask >( y, s, A, x,
mySemiring );
```

3.4 Element-wise Lambdas

Yzelman et al. recognized that in the *blocking* mode of execution where every primitive call must complete before returning [BMM+17], performance may suffer due to unnecessary data movement in memory-bound computations. Take the following code for example.

```
grb::operators::min< double > minOp;
grb::eWiseApply( y, x, x, minOp );
grb::eWiseApply( x, y, y, addOp );
```

If the two primitives are executed one by one, elements from x and y are brought from the main memory to the CPU core(s) twice[2]. However, this could be reduced to once if the two calls were fused. Hence, Yzelman et al. introduced the `grb::eWiseLambda` primitive that allows the execution of arbitrary lambda functions on one or more vector containers. The above snippet, for example, could be replaced with a semantical equivalent as below:

```
grb::operators::min< double > minOp;
grb::eWiseLambda( [&x, &y] (const size_t i) {
        grb::apply( y[ i ], x[ i ], x[ i ], minOp );
        grb::apply( x[ i ], y[ i ], y[ i ], addOp );
    }, x, y
);
```

This fuses the calls to y and x and may complete up to 2× faster than the preceding code.

The square bracket vector operator (e.g., $x[\ i\ ]$) in ALP is only valid when 1) it is used inside a lambda function passed to `grb::eWiseLambda`, and 2) index i refers to a nonzero value that was already present in the related container when the eWiseLambda was invoked. Any other use invites undefined behavior. The first vector argument to `grb::eWiseLambda` (x on the second-to-last line) defines which indices the eWiseLambda shall iterate over. The other vector arguments (such as y on that same line) must correspond to any vectors that are captured in the lambda. For example, the following snippet will only set $x_{n/2}$ equal to 3.14, while leaving any other non-zero value of x unmodified:

```
grb::eWiseLambda( [&x] (const size_t i) {
        x[ i ] = 3.14;
    }, s, x
);
```

This element-wise lambda, apart from its intended use case of manually fusing ALP operations, provides critical functionality for translating vertex-centric programs into ALP (demonstrated in Section 5.4). Moreover, in the context of humble programming, recent work by Mastoras et al. provides mechanisms by which ALP/GraphBLAS may automatically fuse operations [MAY23, MAY22] — Section 7.2 discusses the implications of this recent development for vertex-centric programs.

3.5 Final Remarks

All ALP primitives return error codes, which this paper does not discuss for the sake of brevity. A special note, however, is that the program must guarantee sufficient capacities in output containers, as otherwise the related operation may fail. For example,

```
grb::set( s, false );
```

may fail because the requested capacity of s during construction was 1, which is smaller than its total size n. This work assumes, and in implementation ensures, sufficient vector capacities.

While this section has not introduced the ALP/GraphBLAS API or other ALP extensions in full, it lays the foundation for the remainder of this paper. For full details of the ALP/GraphBLAS API and other ALP extensions, read the papers [BMM+17, YDNNS20, MAY23, MAY22].

4 Motivation

The ALP paradigm expresses programs as sequential, data-centric, and standard C++ programs. It also automatically manages performance and takes care of the corresponding code optimizations and parallelization. For example, ALP enables distributed-memory parallel computations that

[2] This assumes a large enough n compared to the last-level cache of the target architecture.

process graphs of up to 42.5 billion edges over multiple multi-socket nodes using simple humble algorithms. This is made possible by the basic algebraic concepts with which ALP programmers annotate their programs. ALP makes use of those high-level annotations to select the appropriate optimizations automatically, allowing programmers to focus entirely on the mathematics [YDNNS20].

However, while most programmers have studied linear algebra at some point during their studies, most programmers do not consciously use linear algebraic concepts — such as monoids and semirings — on a daily basis. This limits the humble appeal of ALP. Separately, a key question is how many practical workloads naturally map to the ALP paradigm.

In motivating the broader use of ALP and other humble programming paradigms, this section identifies three broad approaches:

- educate programmers on the use of algebraic concepts in programming;
- extend ALP functionalities to support broader ranges of workloads; and
- expand other humble programming models into ALP.

This paper primarily focuses on the third approach.

4.1 Educate

No programming model or language has been successful without educating programmers on its use. In the case of ALP/GraphBLAS, education may be relatively simple because most programmers already make implicit use of algebraic concepts, for example, linear algebra and its implicit use of the real semiring, or set algebraic concepts like orders implicitly used when sorting. Moreover, the idea that programming should closely follow mathematical concepts is not new; it forms the basis behind the design of the STL and generic programming [DS00, SM09, SR14] — standard tools and languages used by a significant portion of programmers. Algebraic concepts also appear in standard computer science texts, with the earliest references to the explicit use of semirings in programming included in the seminal works by Aho, Hopcroft, and Ullman [AHU74] and Cormen, Leiserson, and Rivest [CLR92].

4.2 Extend

Ongoing research should push the boundaries of ALP applicability beyond linear algebra, graph computing, and big data [KG11, KJ18]. This paper already makes use of extensions over the C GraphBLAS standard [BMM+17, BBM+19] introduced by Yzelman et al. [YDNNS20], while Section 7.3 proposes two other extensions that would enable automatic translation of MapReduce programs into ALP. Similar extensions are set to enlarge the scope of ALP applicability and are the subject of ongoing research. It is important that such extensions remain minimal as well as faithful, that is, the core set of ALP primitives should remain as few as possible, while ALP containers, structures, and primitives must have clear corresponding concepts in mathematics.

4.3 Expand

However well programmers may prove to be in handling algebraic concepts explicitly, and however broad the scope of applicability of the ALP paradigm may prove to become, the humble mindset dictates that programmers should always be allowed to use the tools that *they* consider most intuitively suitable for different programming tasks. Nevertheless, unchecked growth in software stacks supporting different programming paradigms, many architectures, and many heterogeneous configurations, does not scale. This implies that the number of programming tools with distinct software stacks should be minimized.

Breaking the paradox of these seemingly conflicting demands, this paper proposes that humble programming models can be automatically expanded into other, more *fundamental* humble programming models. With such a mindset, many humble programming models could exist, though only very few should be fundamental. The many humble models should automatically translate to a fundamental one, and only fundamental programming models should be backed by an automatically optimizing, parallelizing, and ideally architecture-portable software stack.

This paper prototypes this vision by automatically translating the successful, scalable, and humble vertex-centric programming model Pregel [MAB+10] into the ALP paradigm. It shows how automatic translation enables multiple humble programming paradigms that share the same software stack, demonstrates both the performance and scalability of this approach, and consequently proposes ALP as a fundamental humble programming model. With this solution, humble programmers can rely on multiple programming interfaces and select the most suitable ones for each job, while hero programmers can focus their efforts on a limited number of fundamental programming models and software stacks.

5 ALP/Pregel

This section first describes the programming interface for a vertex-centric programming model on top of ALP/GraphBLAS. It is a pure vertex-centric interface that does not expose ALP concepts, except for a commutative monoid over which incoming messages are to be reduced. Using this interface, the SCC algorithm is revisited and precisely formulated, and an ALP/Pregel program is introduced for web page ranking based on the canonical PageRank algorithm [PBMW99].

5.1 Interface

The C++ interface supports two termination mechanisms: vote-to-halt and inactivation of vertex programs. For the former, all vertex programs vote on whether to halt the program, which is terminated only if all active vertices vote to halt it. For the latter, vertex programs can set themselves inactive, after which point they shall no longer participate in subsequent rounds. If all vertex programs are inactive, then the overall program terminates.

Algorithm 1 shows the SCC algorithm in our vertex-centric API, and Algorithm 2 shows a simplified PageRank algorithm. Both algorithms are located in the `grb::algorithms::pregel` namespace and provide a concise way of introducing the ALP/Pregel interface.

For both examples, the vertex-centric program corresponds to the `program` static member function defined. Each program 1) operates on given vertex data of a program-defined type, 2) expects an incoming message of a potentially different type, 3) generates an outgoing message of a potentially third different type, 4) has read access to program-specific data and other parameters, and 5) has access to a predefined PregelState type instance providing read access to Pregel metadata and write access to Pregel termination controls.

5.2 SCC Algorithm

Briefly summarizing the earlier SCC example for undirected graphs, every vertex is initially assigned a unique identifier (ID) integer. Each vertex then broadcasts its ID, overwriting its local ID by the maximum ID received. If the current ID is already the maximum and no update is performed, the program votes to halt the execution. One may intuitively find that, in line with being a humble programming model, this algorithm indeed converges to a correct solution where

Algorithm 1 Vertex-centric SCC algorithm

```
template< typename VertexIDType >
struct ConnectedComponents {

        struct Data {};

        static void program(
                VertexIDType &current_max_ID,
                const VertexIDType &incoming_message,
                VertexIDType &outgoing_message,
                const Data &parameters,
                grb::interfaces::PregelState &pregel
        ) {
                if( pregel.round > 0 ) {
                        if( pregel.indegree == 0 ) {
                                pregel.voteToHalt = true;
                        } else if( current_max_ID < incoming_message ) {
                                current_max_ID = incoming_message;
                        } else {
                                pregel.voteToHalt = true;
                        }
                }
                if( pregel.outdegree > 0 ) {
                        outgoing_message = current_max_ID;
                } else {
                        pregel.voteToHalt = true;
                }
        }

};
```

every component is assigned a unique ID that corresponds to the maximum of values initially assigned to all vertices in the component.

The ID type may be any integer, such as **unsigned int** or **size_t**, and incoming and outgoing messages are of the same type. The algorithm does not require any algorithm-specific parameters (hence the empty Data class on line 4 of Algorithm 1). It exemplifies the use of the in-degree (line 2 of the program body) and out-degree (line 10 of the program body) metadata that our Pregel API provides, and makes use of the vote-to-halt mechanism. The program terminates within d steps, where d is the maximum diameter of all components in G.

Execution of the strongly connected algorithm requires the Pregel interface to be initialized over a specific graph. Conversely, in ALP/GraphBLAS, a graph file is typically opened using a parser and then ingested into a `grb::Matrix` instance, for example, by

```
grb::Matrix< void > A( parser.m(), parser.n(), parser.
nz() );
grb::buildMatrixUnique( A, parser.begin(), parser.end()
);
```

The same graph file should now be ingested into a Pregel instance:

```
grb::interfaces::Pregel< void > pregel(
        parser.n(), parser.m(),
        parser.begin(), parser.end()
);
```

The **void** template parameter to the Pregel interface indicates that edge weights must be ignored. This is because Pregel algorithms determine the message patterns based on the edge structures. Nevertheless, the template argument remains for future extensions that may be considered edge-centric, or vertex- and edge-centric, programming paradigms.

Once the Pregel instance is instantiated, it may execute any Pregel algorithm on the underlying graph. The execution employs the execute member function of the Pregel

Algorithm 2 Vertex-centric PageRank algorithm

```
template< typename IOType >
struct PageRank {

        struct Data {
                IOType alpha = 0.15;
                IOType tolerance = 0.00001;
        };

        static void program(
                IOType &current_score,
                const IOType &incoming_message,
                IOType &outgoing_message,
                const Data &parameters,
                grb::interfaces::PregelState &pregel
        ) {
                // initialise
                if( pregel.round == 0 ) {
                        current_score = static_cast< IOType >(1) /
                                static_cast< IOType >(pregel.num_vertices);
                }

                // compute
                if( pregel.round > 0 ) {
                        const IOType old_score = current_score;
                        current_score = parameters.alpha +
                                ( static_cast< IOType >(1) - parameters.alpha ) * incoming_message;
                        if( fabs(current_score-old_score) < parameters.tolerance ) {
                                pregel.active = false;
                        }
                }

                // broadcast
                if( pregel.outdegree > 0 ) {
                        outgoing_message = current_score / static_cast< IOType >(pregel.outdegree);
                }
        }
};
```

instance. For example, in the case of starting Algorithm 1,

```
const size_t n = pregel.num_vertices();
const size_t max_steps = n;
size_t steps_taken;
grb::Vector< size_t > component_IDs( n ), in_msgs( n ),
out_msgs( n );
grb::RC error_code = pregel.template execute<
        grb::operators::max< VertexIDType >,
        grb::identities::negative_infinity
> (
       &(grb::algorithms::pregel::ConnectedComponents::
program),
       component_IDs,
         grb::algorithms::pregel::ConnectedComponents::
Data(),
       in_msgs, out_msgs,
       steps_taken, max_steps
);
```

the execute function is a templated member function whose template arguments define the commutative monoid under which incoming messages are aggregated. Its domains must match that of the outgoing and incoming messages. For SCC, all domains should be of the same integer type. For a specific vertex, using a monoid structure rather than an arbitrary message combiner operator helps ensure the following:

- In cases when no incoming messages are received (e.g., because a vertex in-degree is zero or all vertices with edges incident to this vertex have become inactive), the monoid identity can be substituted as a received message, thus ensuring consistent behavior while monoid associativity ensures scalable reduction.
- In cases when multiple messages are received in an order that potentially differs across rounds and executions, the commutativity of the combiner monoid ensures consistent behavior.

In the SCC example, we selected the maximum component ID. As such, Algorithm 1 demonstrates how the max-monoid for message aggregation is passed to the executor. The non-template arguments include, in order: 1) the vertex-centric program, 2) the vertex states as a dense vector, 3) the program parameters, 4) buffers for the incoming and outgoing messages, and 5) an output field that records the number of rounds the program took before termination. An optional argument, max_steps, limits this number of rounds. If max_steps is not provided, the program will run until a termination condition arises.

The error_code `grb::SUCCESS` will be returned if the algorithm terminates correctly, and `grb::FAILED` will be returned if the algorithm did not reach a termination condition after the maximum number of rounds was completed. `grb::MISMATCH` will be returned if the message buffers do not match the size of the number of vertices in the underlying graph, and `grb::ILLEGAL` will be returned if any of the passed vectors do not have full capacity.

5.3 Vertex-centric PageRank Algorithm

In the second example, the vertex-centric PageRank in Algorithm 2 is simplified to the point where it no longer corresponds to the canonical algorithm [PBMW99]. Despite this, it is a common approximation that appears in vertex-centric and Spark-based literature, for example, Gonzalez et al. [GXD+14]. The algorithm has two canonical algorithm parameters: α and *tol*. The former, α, can be viewed as the regularization parameter, or more intuitively, the probability that someone viewing a web page visits another web page at random rather than via a link on the current page. The latter, *tol*, signifies a desired local tolerance. Once it is met, the current vertex will be set to inactive.

The local state of every vertex is its PageRank score, represented here by the value of IOType (e.g., **double**). In each round, the vertex-centric program distributes the local score equally to all neighbors of the current node. Specifically, the local score s is divided across all neighboring vertices as an outgoing message with value s/d, where d is the out-degree. Incoming messages are aggregated via the standard additive monoid, resulting in an incoming score of s'. Finally, the vertex-centric program determines the new local score as $\alpha + (1-\alpha)s'$. If the difference with the previous local score is less than *tol*, the program considers the local vertex converged and removes it from further rounds of computation. In this case, the local vertex's active neighboring nodes assume that the now-inactive vertex keeps sending the aggregator monoid identity as its outgoing message.

In this example, the out-degree of each vertex is used in computing an outgoing message. This example demonstrates the use of the `PregelState::round` field — used to keep track of the current round of computation — to initialize the vertex weights during the first round (lines 1–5 of the program body). Furthermore, this example demonstrates that in initialization, global information on the total number of vertices in the program (`PregelState::num_vertices`) can be employed within vertex-centric programs by normalizing the initial PageRank score[3]. The inner `if`-statement of the compute block shows how a vertex removes itself from computation. And, in this example, no novel concepts are introduced as to the

[3] This normalization is didactic – it is not necessary for this PageRank-like algorithm to converge.

mechanisms used to broadcast outgoing messages (lines 17–20 of the program body).

As shown in the preceding example, the PageRank program is executed on a Pregel instance. However, the vertex weights and messages are of type **double** instead of **size_t**, and a regular additive monoid is used instead of a max monoid. As per convention, ALP/Pregel algorithms also provide a static member `function::execute` that constructs the necessary communication buffers and monoid definition. With a Pregel instance constructed over some input graph as before, the following executes the ALP/Pregel PageRank algorithm using the default algorithm parameters and an unlimited number of rounds:

```
grb::Vector< double > pr_scores( n );
grb::set( pr_scores, 0 );
grb::algorithms::pregel::PageRank< double >::execute(
     pregel, pr_scores, rounds_taken
);
```

Theoretically, termination of this program is not guaranteed, especially when rounds are limited to a small number or when a low α is chosen [LM11]. Section 6 takes care to report only experiments where program executions terminate successfully.

As noted earlier, this PageRank algorithm does not correspond to the canonical version by Page et al. in that it lacks contributions from the dangling nodes[4]. A common extension to the original Pregel framework [MAB+10] adds global aggregation mechanisms, which would allow for correct implementation of the PageRank algorithm. The addition of these mechanisms corrects the PageRank updates with contributions from the dangling nodes and enables global convergence detection. Such mechanisms are provided, for example, by the Giraph [Ave11] vertex-centric framework. ALP/Pregel, however, does not provide such mechanisms, as the author believes they will limit the potential of automatic overlap of message exchanges with the execution of rounds. This is discussed in more detail in Section 7.2.

ALP/Pregel supports permanently removing vertices from execution by setting them inactive. Although this is a known optimization for improving convergence for the PageRank algorithm [LM11], it is not supported by all vertex-centric frameworks. While this mechanism may accelerate convergence for the PageRank algorithm, ALP/Pregel also exploits inactive vertices to accelerate per-round computations, as described in Section 5.4. The examples thus far have introduced all fields available in `grb::interfaces::PregelState`, except for `num_edges`.

[4] Nodes with out-degree zero.

Table 1 summarizes all available fields.

Table 1 All fields available to a vertex-centric program during computation rounds. The fields are sorted from writable ones that control program termination, to read-only global constants and variables, and finally to read-only constant numbers regarding local vertex properties.

Field name	read or write	Description
active	write	Set to false to become inactive in subsequent rounds
voteToHalt	write	Set to true to vote for halting the program
round	read	The current computation round
num_vertices	read	The total number of vertices in the current graph
num_edges	read	The total number of edges in the current graph
indegree	read	The in-degree of the current vertex
outdegree	read	The out-degree of the current vertex
vertexID	read	The unique ID of the current vertex

5.4 Implementation

The implementation of the vertex-centric ALP/Pregel interface translates the program to a while-loop around standard ALP/GraphBLAS primitives at compilation time. Each execution of the body of the while loop corresponds to the execution of one round of computation as well as the subsequent termination detection and message exchange. Prior to the first round, all vertices are added to the *active list*, and the outgoing message buffer is reset to the monoid identity. During each round, the following operations take place for vertices that are in the active list:

1 reset the outgoing message buffers using the aggregation monoid identity;

2 call the user-defined vertex-centric program, passing in the current vertex state the program operates on, the buffers for incoming and outgoing messages, the algorithm-specific global data (e.g., α and tol for the vertex-centric PageRank), and the PregelState instance;

3 determine whether all active vertices have voted to halt;

4 remove the vertices that have set themselves inactive from the active list;

5 determine whether all vertices have become inactive;

6 increment the round counter;

7 reset the incoming message buffers using the aggregation monoid identity; and

8 exchange messages and aggregate incoming messages via `grb::vxm`.

Steps 5–8 use the updated active list from step 4.

Data structures. The active list, incoming messages, outgoing message, vertex states, in-degrees, out-degrees, and vertex IDs are all maintained as `grb::Vectors` with non-zeroes of the following types: **bool** for the active list, user-defined for the message and state vectors, and **size_t** for the degree and ID vectors. All vector sizes equal the number of vertices in the graph, n. On initialization of a `grb::interfaces::Pregel` instance, the active list is initialized to **true** for all vertices, and the in-degrees and out-degrees are computed using SpMV multiplication of the graph adjacency matrix with a vector of ones. The vertex IDs are computed via

```
grb::set< grb::descriptors::use_index >(
    vertexIDs, 0 );
```

The use_index descriptor writes the index i to each element of vertexIDs, instead of the given scalar value 0.

The values of the outgoing (and incoming) message vector are reset each round via

```
grb::set< grb::descriptors::structural >(
    outgoingMessages, activeList, id
);
```

Here, `id` is the identity of the aggregation monoid, which is of the same type as that of outgoing messages. Note that the preceding code resets only the `outgoingMessages` of active vertices and throws away entries corresponding to inactive vertices.

Calling the user program. The user program executes through a call to `grb::eWiseLambda`. The lambda function passed into the eWiseLambda captures a reference to the active list, incoming and outgoing messages, state, in-degrees and out-degrees, and vertex IDs. It then calls the user-defined program. The eWiseLambda only executes for the vertices that remain active in every round by passing activeList as the leading mask of the element-wise lambda primitive.

Updating the active list. For efficiency, the active list should be maintained as a sparse vector where the number of non-zeroes equals the number of active vertices at the start of each round. All masked operations can then take the structural descriptor, which prevents touching the actual Boolean values of each entry in the active list. However, the user must be allowed to set an active node to **false**, which does require that actual Boolean values be present in the active list. Therefore, the update of active vertices takes place directly after calling the user program, and is implemented using `grb::set(buffer, activeList, true)` *without* passing in a structural descriptor. This operation is the only step primitive in the ALP/Pregel implementation without a structural descriptor. The use of the masked set operation ensures that there are fewer non-zeroes in the buffer. Finally, the buffer is swapped with the active list to be used as the new mask for subsequent steps of the algorithm.

An extra buffer is maintained to support this update step. According to the performance semantics defined by ALP/GraphBLAS, the `grb::set` on a non-empty container implies a cost proportional to the number of non-zeroes before the update, plus a cost proportional to the number of non-zeroes after the update [YDNNS20]. As such, while the buffer takes up $\Theta(n)$ memory, the overhead of this update step is bounded as $\Theta(k)$, with k being the number of active vertices before the update.

Termination detection. Termination detection using the vote-to-halt mechanism takes place via a reduction of the voteToHalt vector into a Boolean scalar using the logical AND monoid, whereas termination detection using the inactive mechanism takes place by counting the number of non-zeroes in the updated active list.

Message exchange. Describing how to perform message exchange and aggregation using vector–matrix multiplication may be unclear if the semiring under which this proceeds is not described. The given aggregation monoid functions as the additive monoid of such a semiring, from whence the requirement that the aggregation monoid needs to be a commutative monoid. The semiring multiplicative operator is

```
grb::operators::left_assign_if<
        IncomingMessageType, bool, IncomingMessageType
>
```

with the Boolean **true** as its identity element. Denoting the application of this multiplicative monoid operation as $x \otimes y = left\_assign\_if(x, y)$, with x from a user-defined domain D and $y \in \{false, true\}$, then $left\_assign\_if : D \times \{false, true\} \to D$ is defined as follows:

$$x \otimes y = left\_assign\_if(x, y) = \begin{cases} x, \text{ if } y \text{ equals } true \\ \mathbf{0}, \text{ otherwise.} \end{cases}$$

In the latter case, **0** corresponds to the additive monoid identity.

The resulting semiring is, in fact, an improper one — because the aggregator identity may be user-defined, it may not annihilate under the multiplicative operator. However, if the additive monoid is the logical OR with **false** as the additive identity, the resulting semiring with *left_assign_if* is, in fact, proper since it reduces to the standard Boolean semiring.

The way we use semiring guarantees that the multiplicative operator is only ever called using the multiplicative identity as the right-hand side input argument, thereby ensuring that improper cases for non-logical-OR aggregators are not triggered. This must be proceeded carefully as follows: The `grb::vxm` operation $in = outG$ matches non-zeroes $g_{ij} \in G$ to corresponding elements out_i from the outgoing message buffer, and first applies the multiplicative operator to form a temporary $tmp = out_i \otimes G_{ij}$. Because G is a **void** matrix, g_{ij} resolves to the semiring identity, **true**, so that

$$tmp = out_i \otimes true = assign\_left\_if(out_i, true) = out_i.$$

This temporary value is then aggregated into in_j using the user-defined aggregator monoid. The ALP/Pregel implementation thus ensures that explicit multiplication with zero (which may not annihilate) never occurs.

5.5 Summary and Costing

The implementation of vertex-centric programming in ALP/GraphBLAS makes ample use of its main features:

- composability of monoids and semirings provides the flexibility to integrate user-defined aggregation into the richer algebraic semiring structure;
- `grb::eWiseLambda` allows the execution of any user-defined operation on the vertex data;
- sparsity, masks, and algebraic structures are exploited to achieve high performance automatically.

The implementation requires one ALP/GraphBLAS vector container for each of the entries in Table 1, as well as one buffer vector for the active list. Through these containers, ALP/Pregel uses $\Theta(n)$ memory for executing any Pregel program. Execution neither allocates nor frees any memory. Computing termination conditions costs $\Theta(k)$ work, with k being the number of active vertices in a given round. In the worst-case scenario, capturing vector elements as arguments to user-defined vertex-centric programs has $\mathcal{O}(1)$ overhead, again translating to $\mathcal{O}(k)$ costs per round. The cost of executing a vector program is determined by the user, and is linear with k. Specifically, for a vertex-centric program that locally takes $\Theta(1)$ time, executing all active programs costs $\Theta(k)$ work.

Again in the worst-case scenario, data movement complexities in a shared-memory setting correspond to $\Theta(k)$ with regard to accessing internal vector states. The persistent state of a single vertex and how much of it is touched during each round is user-defined. However, if we assume $\Theta(1)$ memory movement by a vertex-centric program during any round, the total memory movement is also $\Theta(k)$ per round. Assume that the sizes of incoming and outgoing messages are $\Theta(1)$, the message exchange costs

$$\mathcal{O}\left(k + \sum_{i \in activeList} d_i\right) \quad (1)$$

data movement per round, where d_i is the in-degree of the i-th vertex. Although perhaps counterintuitive, exchanging messages also implies work. For ALP/GraphBLAS, the work bound equals that of Eqn. (1), except that the bound is big-Theta instead of big-Oh.

For shared-memory parallelization, the above work bounds may be divided by T, where T is the number of threads. Parallelization also adds a factor $\mathcal{O}(T)$ to all storage, work, and data movement bounds. Consequently, the work bound corresponds to the per-thread work, while data movement bound considers the data volume moved across the whole system.

Likewise, for distributed-memory parallelization, work can be divided by P, which is the number of distributed-memory processes, while adding a factor $\mathcal{O}(P)$ to all storage, work, and data movement bounds. Additionally, the active list update and the vote-to-halt termination check each amount to a collective reduction across all nodes. This induces $\mathcal{O}(P)$ inter-process data movement and work, and as many as $\mathcal{O}(\log P)$ synchronization steps, where P is the number of distributed processes. The message exchange adds $\mathcal{O}(k)$ inter-process data movement, assuming an $\Theta(1)$ storage for the outgoing messages. Tables 2 and 3 summarize all costs.

These per-round costings are a direct application of the ALP/GraphBLAS performance semantics [YDNNS20]. They confirm that, for an increasing number of inactive vertices, bounds for work and both intra- and inter-process data movement improve.

Table 2 Costs of executing a single round of an ALP/Pregel program on a shared-memory machine. This assumes there are n vertices, of which k are active, m edges, $\Theta(1)$ execution time of a vertex-centric program, and $\Theta(1)$ storage for each of a single vertex state, incoming message, and outgoing message. The work corresponds to that of a single thread, and the data movement corresponds to that across the entire machine. The sum $\sum_i d_i$ is as in Eqn. (1), while T indicates the number of threads within a shared-memory machine. Sequential costs correspond to $T = 1$.

	Storage	Work	Data movement
Round computation	$\Theta(n+T-1)$	$\Theta(k/T+T-1)$	$\Theta(k+T-1)$
Active list update	$\Theta(n+T-1)$	$\Theta(k/T+T-1)$	$\Theta(k+T-1)$
Termination check	$\Theta(T)$	$\mathcal{O}(k/T+T-1)$	$\mathcal{O}(k+T-1)$
Message exchange	$\Theta(n+m+T-1)$	$\Theta((k+\sum_i d_i)/T+T-1)$	$\mathcal{O}(k+[\sum_i d_i]+T-1)$
Total	$\Theta(n+m+T-1)$	$\Theta((k+\sum_i d_i)/T+T-1)$	$\mathcal{O}(k+[\sum_i d_i]+T-1)$

Table 3 Additional costs of executing a single round of an ALP/Pregel program on a distributed-memory architecture. The costs are in addition to those listed in Table 2, only that T should be replaced with the number of threads available at each process multiplied by the number of distributed-memory processes P. Additionally, all its data movement costings should be divided by P, and each of storage, work, and data movement adds a factor $\mathcal{O}(P)$.

	Work	Data movement	Synchronizations
Round computation	0	0	0
Active list update	$\mathcal{O}(P)$	$\mathcal{O}(P)$	$\mathcal{O}(\log P)$
Termination check	$\mathcal{O}(P)$	$\mathcal{O}(P)$	$\mathcal{O}(\log P)$
Message exchange	$\Theta(k)$	$\Theta(k)$	1
Total	$\Theta(k)+\mathcal{O}(P)$	$\Theta(k)+\mathcal{O}(P)$	$\mathcal{O}(\log P)$

6 Experiments

Experiments are conducted for two purposes: 1) to show that humble ALP/Pregel programs can achieve the scalability predicted in the previous section, and 2) to show that ALP/Pregel is competitive against state-of-the-art frameworks. For the first purpose, the SCC algorithm is investigated, comparing the sequential and parallel results. Then, using two variants of the PageRank algorithm, the behavior of ALP/Pregel under an increasing number of inactive vertices is investigated. Finally, a scalability experiment using PageRank algorithms demonstrates that shared-memory auto-parallelization of ALP/Pregel behaves as expected also when the number of inactive vertices increases.

For the second purpose of comparing ALP/Pregel with state-of-the-art frameworks, this section first summarizes the ALP/GraphBLAS performance from the recent work by Mastoras et al. [MAY23], who provide an in-depth comparison of ALP/GraphBLAS against the GNU Scientific Library (GSL), Eigen [GJ+10], and SuiteSparse:GraphBLAS [Dav19]. They show that the ALP/GraphBLAS PageRank implementation sketched in Algorithm 3 is 0.96–9.82 times faster than a PageRank implemented in SuiteSparse:GraphBLAS, and 0.64–14 times faster than one written using Eigen on shared-memory parallel architectures. Both SuiteSparse and Eigen auto-parallelize over shared-memory architectures, while GSL does not. Eigen additionally performs loop fusion, which leads to speedups over the blocking ALP/GraphBLAS for small matrices. The nonblocking ALP/GraphBLAS implementation that Mastoras et al. contribute also achieves loop fusion and achieves speedups beyond Eigen for both smaller and larger matrices, but will not be considered as part of the experiments in this section. Mastoras et al. obtain similar results for two other sparse matrix and graph algorithms.

This paper focuses on comparing the ALP/Pregel PageRank-like algorithm with the canonical ALP/GraphBLAS variant, which is taken as the state-of-the-art baseline. This paper does not compare ALP/Pregel against the existing vertex-centric frameworks such as Giraph [Ave11], since such frameworks typically exhibit orders-of-magnitude performance losses versus the optimized code [SY19] because they rely on file-based fault tolerance. Such comparisons are out of our scope because what we look for are humble programming models that achieve hero-level performance.

6.1 Methodology

Experiments are run on a dual-socket Intel x86 machine, consisting of two Intel Xeon 6238T processors, each having 22 cores, a 32 kB private L1 cache per core, a 1 MB private L2 cache per core, and a 30.25 MB shared L3 cache. The cores, clocked at 1.9 GHz, have hyperthreading enabled and turbo boost disabled. Each processor has six memory channels clocked at 2,933 MT/s, producing 262.2 GB/s of theoretical throughput across the total machine. The combined computational throughput of all AVX-512 enabled cores is 2,675 Gflop/s. The single-core memory throughput ranges from 9.95 (vector-to-scalar reduction) to 18.4 (triad) Gbyte/s. These figures are relevant because ALP/Pregel computations are typically memory-bound, achieving far lower speedup than the total number of cores would indicate. Indeed, for this architecture, a bandwidth-bound application is expected to achieve 14.3–26.4 times of speedup.

Our experiments are implemented on v0.6 of ALP/GraphBLAS, which is available on GitHub and Gitee [Alg21a, Alg21b]. Its sequential reference and shared-memory parallel reference_omp backends are used both to compile ALP/Pregel programs and to provide a baseline PageRank implementation. All software is compiled using GCC 9.3.1 with the **-O3 -mtune=native-march=native -DNDEBUG -funroll-loops** compiler flags. All compilations and experiments are executed on Linux kernel version 5.8.18. Experiments using OpenMP define **OMP_PROC_BIND=true**.

Experiments on small datasets such as gyro_m are repeated multiple times to ensure that one timing takes at least 100 milliseconds. Then, experiments are further repeated at least 10 times in order to compute the sample standard deviation across all timings. All reported timings have a sample standard deviation of less than 3% of the measured average time. Timings with sample standard deviation being 1% higher than the average time are printed in *italics*, and the tables only show three significant digits of each of the obtained averages. As such, timings printed in non-italics are accurate to a significant degree, while for timings printed in italics the least significant digit is likely to be inaccurate.

In this paper, the presented methodology is only not applied to three experiments benchmarking the sequential ALP/Pregel SCC due to a very long run time.

6.2 Datasets

Our experiments require input graphs on which to run the vertex-centric algorithms. Typical datasets that vertex-centric frameworks operate on include knowledge graphs and graph representations from the Web and other types of networks. To also compare against structures that differ significantly from typical graphs so that we can gauge the effectiveness when faced with problems originating from other domains, we include both graphs and sparse matrices from various scientific computing domains in our dataset, as shown in Table 4.

Table 4 Graphs used in the experiments. All except 11 and 12 are undirected.

Dataset	Name	#Vertices	#Edges	Edges/Vertex	Structured
1	gyro_m	17 361	340 431	19.6	no
2	vanbody	47 072	2 336 898	49.6	yes
3	G2_circuit	150 102	726 674	4.84	mixed
4	bundle_adj	513 351	20 208 051	39.4	adverse
5	apache2	715 176	4 817 870	6.74	yes
6	ecology2	999 999	4 995 991	5.00	yes
7	Emilia_923	923 136	41 005 206	44.4	yes
8	Serena	1 391 349	64 531 701	46.4	yes
9	G3_circuit	1 585 478	7 660 826	4.83	mixed
10	Queen_4147	4 147 110	329 499 284	79.5	yes
11	wikipedia-20070206	3 566 907	45 030 389	12.6	no
12	uk-2002	18 520 486	298 113 762	16.1	no
13	road_usa	23 947 347	57 708 624	2.41	no

All datasets are from the SuiteSparse MatrixMarket collection [DH11], and are ordered by the number of vertices they contain. In the architecture described earlier, 128k 64-bit words (**double**s or **size_t**s) fit into private L2 caches. As such, we expect significant data reuse by algorithms operating on datasets 1 and 2. Approximately 4M words fit into shared L3 caches. Given that all algorithms require multiple vectors to operate, significant reuse of data in L3 caches should occur for datasets 3–10. Datasets above number 10 will always induce out-of-cache behavior because the corresponding vectors do not fit in L3 caches. As we will describe later in this section, performance differences between some PageRank implementations results from the difference in the number of edges per vertex (also reported in Table 4).

Apart from differences in the number of vertices and edges, we augment graphs that have no discernible structure when plotting its corresponding adjacency matrix with structured matrices. Discernible structures in such matrices include having a fixed number of diagonals in the adjacency matrix, and with only non-zeroes within a fixed bandwidth around the main diagonal. Structured matrices and graphs with structured corresponding adjacency matrices induce favorable data access patterns that modern hardware prefetchers implicitly exploit, which results in very different behavior during computations on such graphs. These effects are well-known in high-performance computing research that deals with sparse matrices. Some datasets, such as G3_Circuit, have both structured and unstructured components in the adjacency matrix, whereas bundle_adj has an adversarial structure that has a major impact on the performance of linear algebraic libraries and programming frameworks [MAY23]. All graphs represented by the adjacency matrices are undirected, except wikipedia-20070206 and uk-2002.

6.3 SCC Algorithm

We first consider the scalability of ALP/Pregel using the SCC for undirected graphs in Algorithm 1. For directed graphs this algorithm still terminates, and returns connected components where for each vertex v in a given component, there exists at least one other vertex u in that same component with a path to u to v. Given Algorithm 1 over directed inputs retains a meaningful interpretation, our experiments use the full dataset in Table 4.

Table 5 compares the wall-clock time of executing the related ALP/Pregel SCC algorithm between two modes: using the sequential ALP/GraphBLAS backend and using the shared-memory parallel OpenMP-enabled backend. We emphasize that parallelization is fully automatic and the code presented in Algorithm 1 does not need to be changed.

Table 5 The runtime in milliseconds for executing the ALP/Pregel SCC, Algorithm 1, compiled using the sequential and shared-memory parallel ALP/GraphBLAS backends. The numbers of iterations are listed in the parentheses following each timing, and the last column reports the speedup of parallel execution over sequential execution.

Dataset	Sequential	Shmem. parallel	Speedup
1	39.3 (47)	38.6 (47)	1.03×
2	183 (53)	56.2 (53)	3.26×
3	755 (162)	255 (162)	2.96×
4	4 910 (140)	568 (140)	8.64×
5	11 100 (447)	1 970 (447)	5.63×
6	52 300 (2000)	11 400 (2000)	4.59×
7	*6 930* (111)	774 (111)	8.95×
8	*6 090* (59)	618 (59)	9.84×
9	27 400 (523)	4 500 (523)	6.09×
10	*76 200* (178)	5 620 (178)	13.6×
11	189 000 (461)	10 800 (461)	17.5×
12	*131 000* (116)	11 900 (116)	11.0×
13	6 080 000 (5681)	668 000 (5681)	9.10×

The number of rounds required by the algorithm is shown in the parentheses. In addition, the table also reports speedup obtained by shared-memory parallelization.

The SCC algorithm employs only the vote-to-halt termination mechanism so that each round runs with all vertices being active. Moreover, parallelization should not affect the number of rounds computation requires. This experiment therefore measures only the scalability of the shared-memory parallel backend using the SCC algorithm.

Parallelization involves unavoidable overheads of at least $\Omega(\log T)$. In ALP/GraphBLAS, it is bounded by $\mathcal{O}(T)$ [YDNNS20]. Therefore, it is certain that for smaller datasets on a full system with 88 hyperthreads and two NUMA domains, speedups are expected to be far below 26.4x, which is achieved by the triad benchmarks[5]. Since the vote-to-halt termination mechanism depends on the vector-to-scalar reduction, a significant portion of the vertex-centric paradigm will be bound by the 14.3x of speedup for the reduction benchmark[5]. However, as the size of datasets increases, speedups are expected to reach the 14.3–26.4 range.

Table 5 reports the same number of rounds for both the sequential and shared-memory parallel ALP/Pregel SCC algorithms. The maximum speedup, 17.5x for dataset 11, falls within the upper range, indicating that parallelization of the non-reduction parts of the ALP/Pregel program achieves speedups closer to the best-case triad benchmark. This indicates that the ALP/Pregel implementation scales on shared-memory architectures for the particular case when all vertices remain active throughout the computation.

6.4 Sequential PageRank

By contrast, the PageRank in Algorithm 2 does make use of the inactive vertices. In this case, as the computation proceeds, fewer and fewer vertices will partake in the computation. Termination of this program does not guarantee convergence in the classical 2–norm, nor in the inf-norm, even if dangling nodes were accounted for properly.

To compare the impact of the vertex inactivation mechanism on the speed of computation, we introduce a global variant of the ALP/Pregel PageRank by modifying Algorithm 2. Specifically, instead of using the inactivation mechanism, the algorithm is modified to use vote-to-halt on local convergence detection. As such, we subsequently refer to the original Algorithm 2 as the local variant, because it does not consider a global inf-norm computation but a greedy local version of it.

Both the global and local ALP/Pregel PageRank algorithms are further compared against the standard PageRank implementation using and bundled with ALP/GraphBLAS. This algorithm does account for dangling nodes and implements convergence detection in the standard 2–norm [YDNNS20]. Algorithm 3 provides a simplified listing of that algorithm for the sake of completeness[6]. Because of the algorithmic differences relating to dangling nodes, the standard equivalence of norms does not apply, and we therefore cannot directly compare errors between the ALP/Pregel implementations and the ALP/GraphBLAS implementation (the algorithms are different and converge to different values). As described in Section 5.3, the vertex-centric PageRank is commonly applied, and its global variant is most often applied. Table 6 therefore compares the local vertex-centric variant against its global variant in terms of performance and speed of convergence. It also compares both Pregel variants and the canonical PageRank approaches to web page ranking in terms of performance.

In terms of the number of iterations, among the three different algorithms we compare, it is not the case that the vertex-centric algorithms always incur fewer iterations than the canonical PageRank, nor that the local variant of the ALP/Pregel PageRank always incurs fewer iterations than its global variant. Indeed, as Table 6 shows, any algorithm can incur the fewest number of iterations, depending on the input graphs. The vertex-centric variants do require significantly more iterations for the large undirected graphs, IDs 11 and 12, to converge, but this may well be due to the common presence of dangling nodes in those graphs.

In terms of the execution speed, the local variant is faster than the global variant. This is because, as the rounds progress, increasingly more vertices become inactive. This is confirmed by the time per iteration reported in Table 6. As such, we may conclude that our ALP/Pregel implementation becomes faster as the number of active nodes decreases. One disadvantage of using the element-wise lambda noted by Yzelman et al. is that the compiler is no longer able to apply vectorization when possible [YDNNS20]. Thus, the ALP/GraphBLAS PageRank may be faster than the global ALP/Pregel variant even though the former performs more operations[7]. This benefit is expected to decrease as the number of edges per vertex increases.

[5] See Section 6.1.

[6] See the ALP/GraphBLAS repository [Alg21a] for the full algorithm.

[7] i.e., dangling node corrections and 2–norm computations.

Table 6 Results, in milliseconds, of executing sequential PageRank algorithms, comparing the ALP/Pregel variants using global and local convergence against the baseline ALP/GraphBLAS PageRank implementation. The numbers of iterations implemented before convergence are listed in the parentheses. The last three columns report the time per iteration, with the fastest marked in **bold**.

ALP/Pregel				ms. per iteration		
Dataset	Global	Local	Baseline	Global	Local	Baseline
1	34.8 (40)	24.7 (39)	31.4 (52)	0.870	0.633	**0.604**
2	192 (43)	118 (41)	197 (52)	4.47	**2.88**	3.79
3	175 (38)	78.8 (36)	90.0 (48)	4.61	2.19	**1.88**
4	3 070 (66)	2 070 (51)	2 330 (60)	46.5	40.6	**38.8**
5	*707* (31)	456 (33)	*434* (43)	22.8	13.8	**10.1**
6	976 (31)	63.5 (34)	*375* (30)	31.5	**1.87**	12.5
7	2 840 (36)	1 180 (36)	*2 960* (45)	78.9	**32.8**	65.8
8	5 090 (40)	*2 500* (33)	4 750 (44)	127	**75.8**	108
9	1 960 (38)	987 (36)	*1 100* (48)	51.6	27.4	**22.9**
10	20 800 (35)	*2 780* (35)	25 000 (46)	594	**79.4**	543
11	40 500 (103)	11 400 (96)	18 100 (55)	393	**119**	329
12	153 000 (115)	*46 100* (104)	*72 100* (73)	1330	**443**	988
13	*87 600* (78)	58 800 (72)	62 200 (78)	1120	817	**797**

The time per iteration reported in the preceding table shows that the ALP/GraphBLAS outperforms the global variant, and even the local vertex-centric variant in 6 out of 13 cases. For the larger graphs, the latter is explained by the observation that low edge count per vertex favors vectorization (datasets 6, 9, and 13). Additionally, a performance gain for the pure ALP/GraphBLAS algorithm on small graphs is observed. This gain increases on the shared-memory parallel ALP/GraphBLAS variant, as presented and analyzed in more detail in the next section.

In summary, in terms of the time per iteration, the local ALP/Pregel PageRank is up to 16.8x faster than its global variant, and up to 6.84x faster than a state-of-the-art canonical PageRank algorithm. In terms of the end-to-end runtime, the local variant is up to 7.48x faster than the global one, and up to 8.99x faster than the state-of-the-art baseline — the biggest slowdown is limited at 0.95x. The local ALP/Pregel algorithm is fastest overall in 12 out of 13 cases.

6.5 Shared-memory Parallel PageRank

Similar to the SCC algorithm, in the case of parallelization of PageRank, the number of iterations will not change when the arithmetic, including both its precision and order, remains unchanged. Although numerical error propagation could cause minor variations, comparison of Tables 6 and 7 reveals no difference in the required number of iterations for our algorithms and datasets.

In terms of the execution speed, including both the end-to-end runtime and the time per iteration, there is no instance where the global variant outperforms the local variant. In terms of the end-to-end runtime, the state-of-the-art PageRank implemented directly on top of ALP/GraphBLAS is fastest in eight cases, which are exactly the ones where the canonical PageRank achieves the fastest time per iteration. However, there is one exception where the local variant does achieve the fastest end-to-end runtime, even though in terms of the time per iteration the baseline outperforms the local variant. Indeed, differences in the time per iteration are more pronounced between the state-of-the-art baseline and the local variant, with speedups ranging 0.403–10 times.

Table 8 collects the speedups for all variants, similar to the SCC study in Section 6.3. The speedups ALP/Pregel achieves on bundle_adj (9.24-11.0x) are significantly higher than the speedups achieved by the baseline (1.81x). This is due to the adversarial structure of the graphs, which

groups high-degree vertices in one cluster. Since the ALP/GraphBLAS baseline employs dense unmasked vectors only, the shared-memory parallel ALP backend reverts to static scheduling during SpMV multiplication. The ALP/Pregel variants, however, use a sparse structural mask for which the backend reverts to dynamic scheduling during SpMV multiplication. This, in turn, avoids the imbalance induced by the adversarial graph structure.

The ALP/GraphBLAS PageRank algorithm generally achieves better speedups than the ALP/Pregel variants, and the global variant tends to achieve better speedups than the local variants. While all implementations rely on the same OpenMP-enabled ALP/GraphBLAS backend, they rely on different OpenMP scheduling: as discussed earlier, the ALP/GraphBLAS variant employs static scheduling during sparse matrix–vector multiplication, whereas the ALP/Pregel variants employ dynamic scheduling. Additionally, ALP/Pregel employs dynamic scheduling while executing the vertex-centric program through the element-wise lambda. This is because the user-defined lambda may induce varying workload, depending on the index on which it operates. Moreover, some dense vector–vector operations in the baseline revert to static scheduling, while the same operations in the ALP/Pregel variants are part of the dynamically scheduled element-wise lambda. Some ALP/Pregel metadata updates performed via level-1 operations in the global variant also employ static scheduling, whereas in the local variants they revert to dynamic scheduling.

Another major effect is regarding the workspace of the ALP/Pregel programs. Recall that executing any Pregel program requires eight vectors, whereas the ALP/GraphBLAS PageRank implementation in Algorithm 3 requires only four. This translates to a benefit for smaller graphs, as the baseline implementation induces higher data reuse for the same problems within the same limited caches. This is compounded by an explicit materialization of the messages between vertices in the form of incoming and outgoing messages, meaning that computed values are copied twice as many times as in the baseline implementation, which holds zero such copies.

Algorithm 3 A simplified representation of the ALP/GraphBLAS PageRank

```
using namespace grb;
void pagerank(
    Vector< double > &pr, const Matrix< NonzeroT > &L,
    Vector< double > &temp, Vector< double > &pr_buffer,
    Vector< double > &row_sum,
    const double alpha,
    const double conv
) {
    Monoid< operators::add< double >, identities::zero > addM;
    Semiring< operators::add< double >, operators::mul< double >,
        identities::zero, identities::one > realRing;
    const size_t n = nrows( L );
    set( temp, 1 );
    set( row_sum, 0 );
    vxm< descriptors::dense | descriptors::transpose_matrix >( row_sum, temp, L, realRing );
    eWiseLambda( [ &row_sum, &alpha, &zero ]( const size_t i ) {
            if( row_sum[ i ] > 0 ) { row_sum[ i ] = alpha / row_sum[ i ]; }
        }, row_sum );

    double dangling, residual;
    do {
        residual = dangling = 0;
        foldl< descriptors::invert_mask >( dangling, pr, row_sum, addM );
        set( temp, 0 );
        eWiseApply( temp, pr, row_sum, operators::mul< double >() );
        dangling = ( alpha * dangling + 1 - alpha ) / static_cast< double >( n );
        set( pr_buffer, 0 );
        vxm( pr_buffer, temp, L, realRing );
        foldl< descriptors::dense >( pr_buffer, dangling, addM );
        dot< descriptors::dense >( residual, pr, pr_buffer, addM, operators::abs_diff< double >() );
        if( residual <= conv ) { break; }
        std::swap( pr, pr_buffer );
    } while( true );
}
```

Table 7 Results, in milliseconds, of executing shared-memory parallel PageRank algorithms. Like in Table 6, this table compares the ALP/Pregel variants against the baseline shared-memory parallel ALP/GraphBLAS variant.

ALP/Pregel				ms. per Iteration		
Dataset	**Global**	**Local**	**Baseline**	**Global**	**Local**	**Baseline**
1	31.1 (40)	29.2 (39)	*37.6* (52)	0.778	0.749	**0.723**
2	43.3 (43)	37.0 (41)	*53.8* (52)	1.01	**0.902**	1.03
3	58.8 (38)	38.9 (36)	29.0 (48)	1.55	1.08	**0.604**
4	280 (66)	*224* (51)	1290 (60)	4.24	**4.39**	21.5
5	157 (31)	127 (33)	35.1 (43)	5.06	3.85	**0.816**
6	*203* (31)	*62.3* (34)	32.1 (30)	6.55	1.83	**1.07**
7	263 (36)	163 (36)	*93.0* (45)	7.31	4.53	**2.07**
8	412 (40)	272 (33)	*146* (44)	10.3	8.24	**3.32**
9	367 (38)	243 (36)	87.7 (48)	9.66	6.75	**1.83**
10	1170 (35)	333 (35)	701 (46)	33.4	**9.51**	15.2
11	2440 (103)	878 (96)	5030 (55)	23.7	**9.15**	91.5
12	11500 (115)	4420 (104)	2750 (73)	100	42.5	**37.7**
13	9800 (78)	7560 (72)	2680 (78)	126	105	**34.4**

Table 8 Speedups of the parallel PageRank variants from Table 7 versus the sequential variants from Table 6.

Speedup vs. Itself				Speedup vs. Baseline	
Dataset	**Global**	**Local**	**Baseline**	**Global**	**Local**
1	1.12	0.846	0.835	1.21	**1.29**
2	4.43	3.19	3.66	1.24	**1.45**
3	2.98	2.03	3.10	0.493	0.746
4	11.0	9.24	1.81	4.61	**5.76**
5	4.50	3.59	12.4	0.224	0.276
6	4.81	1.02	11.7	0.158	0.515
7	10.8	7.24	31.8	0.354	0.571
8	12.4	9.19	32.5	0.354	0.537
9	5.34	4.06	12.5	0.239	0.361
10	17.8	8.35	35.7	0.599	**2.11**
11	16.6	13.0	3.60	2.06	**5.73**
12	13.3	10.4	26.2	0.239	0.622
13	8.94	7.78	23.2	0.273	0.354

The switch from static to dynamic scheduling, the loss of data locality, and the explicit materialization of messages result in a worst-case slowdown of 0.340×, comparing the baseline with global variants. Additional uses of dynamic instead of static scheduling in vector–vector and vector–scalar operations compound the worst-case slowdown to 0.212×, comparing the baseline with local variants. The effects of losing data locality should be reduced for larger problems.

Although these slowdowns indicate that ALP/Pregel programs scale worse than pure ALP/GraphBLAS ones, this may be resolved partially by investigating alternative dynamic scheduling techniques within ALP. However, the overheads incurred from copying messages and the loss of data reuse opportunities for small problems remain unavoidable. Nevertheless, the local variant leads to the fastest end-to-end shared-memory parallel execution in 5 out of 13 cases, with speedups of up to 5.76× compared to the baseline.

7 Conclusions

The paramount challenge in contemporary research on programming models and compilers lies in striking a balance between providing easy-to-use programming interfaces that make future highly parallel and heterogeneous compute systems accessible to humble programmers, and the necessity to:

- support an ever-increasing number of architectures,
- deploy over increasingly large-scale systems that involve multiple architectures, and
- deal with a mixture of shared- and distributed-memory connectivity across compute units.

In the interest of exposing humble programming models that are usable by the vast majority of programmers, a hit in performance and scalability may be acceptable. Ideally, however, humble code — once compiled — should perform on par with expert code, thus turning humble programmers into heroes. A recent study shows that the humble ALP/GraphBLAS outperforms the state-of-the-art frameworks for three graph and sparse matrix algorithms [MAY23]. ALP/Pregel exposes a vertex-centric programming model and translates vertex-centric programs into ALP/GraphBLAS at compile time, while offering benefits in terms of its shared- and distributed-memory auto-parallelization and other optimizations. This work demonstrates that a vertex-centric SCC algorithm implemented using ALP/Pregel obtains up to 17.5× speedup on a dual-socket Intel Xeon machine. Similarly, an auto-parallelized vertex-centric PageRank-like algorithm obtains speedups up to 17.8×, confirming that ALP/Pregel programs are able to scale. Furthermore, comparing the vertex-centric program against a highly optimized canonical PageRank algorithm results in faster sequential execution in 12 out of 13 cases, and speedups of up to 8.99×. In shared-memory parallel execution, the ALP/Pregel algorithm is fastest in 5 out of 13 cases, with speedups of up to 5.76×. These results confirm that humble programs may achieve hero-level performance, and demonstrate that novel approaches based on such humble programming interfaces may even outperform highly optimized canonical solutions.

7.1 Outlook

Different groups of humble programmers may prefer different humble programming interfaces. As such, the compute industry and researchers should strive to identify a set of humble programming models that together appeal to the vast majority of programmers. Furthermore, in an ideal scenario, only a few humble programming interfaces with corresponding optimized software stacks would be required. This paradox can be resolved by translating many useful humble programming interfaces into just a few optimized humble software stacks.

The reported speedups from the ALP/Pregel programs are achieved even though those programs are expanded into an ALP program, making use of its standard implementation. This not only validates the notion of supporting multiple humble programming models on top of a single software stack, but also demonstrates that this is possible without significant performance overheads. It also solves the software bottleneck as it frees hero programmers of addressing optimization and portability concerns for the few fundamental software stacks only — in this case, the ALP stack.

Even so, this paper does not argue that ALP should be the only fundamental programming model. It remains an open question as to how much of the general-purpose programming demands can be covered by ALP, as well as how many popular humble programming interfaces can be implemented on top of ALP without incurring significant performance overheads. Instead, this paper humbly argues that the future of software would be vastly improved if the industry and research communities explore more scalable humble programming interfaces, as well as identify the smallest possible subsets of foundational software stacks into which these humble programming models translate.

7.2 Improving ALP/Pregel

Pregel programs are naturally loosely coupled. At any point in time during execution, not all vertices are required to be executing the same round. Specifically, vertices whose round-i messages have been received can immediately continue with round $i+1$, regardless of whether other vertices have yet to execute the ith computation phase or whether they are waiting on incoming round-i messages. This insight can be applied recursively to realize pipelines of arbitrary length, bounded only by the underlying graph structure and the available memory for caching in-flight messages.

For example, a Pregel program running on the source vertex of a line graph can progress arbitrary round numbers ahead of other vertices. This leads to a pipeline depth bounded by the length of the graph. However, a Pregel program running on any vertex of a fully connected graph must execute in lock-step with all other programs due to being connected in an all-to-all fashion. While ALP/Pregel transforms the program into a series of level-1 operations separated by a message exchange driven by an SpMV multiplication, the use of a nonblocking implementation of ALP/GraphBLAS such as by Mastoras et al. [MAY23, MAY22] realizes overlap between the communication and computation phases of a single round. Introducing buffers for incoming messages in ALP/GraphBLAS may likewise enable overlapping rounds of vertex-centric computations. This will be the next step to take to enhance the ALP/Pregel performance. Such overlapping should be particularly effective on distributed-memory architectures, which, although supported by the current ALP/Pregel implementation, has not been evaluated in this work. Such comparison will be included in our future work, in which we will also compare the effects of loosely coupled execution. However, this requires bringing the nonblocking ALP/GraphBLAS capabilities to the distributed-memory parallel case. From the single-node perspective, a nonblocking variant that exploits the underlying graph structure in order to eliminate the need for caching incoming messages — thereby saving memory resources and increasing data reuse — seems a most promising direction.

7.3 Beyond ALP/Pregel

While this paper shows that realizing the Pregel humble programming model on top of ALP is possible and thus may benefit from the high performance and scalability guaranteed by ALP, not all algorithms and workloads are amenable to either pure ALP or vertex-centric programming. We need to expand the range of humble models that we can automatically translate into ALP. For MapReduce, for example, the following insights may automatically translate MapReduce programs into ALP:

- a one-to-one map phase corresponds to the element-wise lambda function;
- one-to-none or one-to-many maps may be realized by the parallel I/O mode introduced by ALP/GraphBLAS [YDNNS20]; and
- a reduce phase may be realized by a four-step process:
  (1) unzipping a key-value ALP/GraphBLAS vector of length n into two vectors of n keys and n values,
  (2) zipping the two vectors into an ALP/GraphBLAS matrix of size $k \times n$,
  (3) performing the reduction via an SpMV multiplication, and
  (4) transforming the resulting vector of values back into a vector of key-value pairs.

This ALP/MapReduce approach is currently under implementation and will be evaluated and released as the next additional humble programming interface to ALP. Inspirations to further humble interfaces on top of ALP include NumPy [ADH+01] and Spark [ZCF+10] as other examples of high-impact humble programming models, as well as programming models based on set algebra [BVSS+21, BKK+21] and commutative sets [KPW+07, PGZ+11].

References

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, *et al.*, "Numerical Python," 2001.

[AHU74] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman, "The design and analysis of computer algorithms," *Reading*, *MA*, 19(4), 1974.

[Alg21a] Algebraic Programming. ALP/GraphBLAS release v0.6. https://github.com/Algebraic-Programming/ALP, 2021. Last retrieved on April 6th, 2023.

[Alg21b] Algebraic Programming. ALP/GraphBLAS release v0.6. https://gitee.com/CSL-ALP/graphblas, 2021. Last retrieved on April 6th, 2023.

[AS87] Baruch Awerbuch and Yossi Shiloach, "New connectivity and MSF algorithms for shuffle-exchange network and PRAM," *IEEE Transactions on Computers*, 36(10):1258–1263, 1987.

[Ave11] Ching Avery, "Giraph: Large-scale graph processing infrastructure on Hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[BBM+19] Benjamin Brock, Aydin Buluç, Timothy Mattson, Scott McMillan, and José Moreira, "The GraphBLAS C API specification," *GraphBLAS.org*, *Tech. Rep*, 2019.

[BKK+21] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pp. 282–297, New York, NY, USA, 2021.

[BMM+17] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium workshops (IPDPSW)*, pp. 643–652. IEEE, 2017.

[BVSS+21] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Marek Konieczny, Onur Mutlu, and Torsten Hoefler, "GraphMineSuite: Enabling high-performance and programmable graph mining algorithms with set algebra," *Proc. VLDB Endow.*, 14(11):1922–1935, Jul. 2021.

[CLR92] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest, "Introduction to Algorithms," MIT Press, first edition, 1992.

[Dav19] Timothy A Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[DG08] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, 51(1):107–113, 2008.

[DH11] Timothy A Davis and Yifan Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[Dij72] Edsger W Dijkstra, "The humble programmer," *Communications of the ACM*, 15(10):859–866, 1972.

[DS00] James C Dehnert and Alexander Stepanov, "Fundamentals of generic programming," *International Seminar on Generic Programming*, pp. 1–11, Springer, 2000.

[GJ+10] Gaël Guennebaud, Benoit Jacob, *et al.*, "Eigen Technical report," TuxFamily, 2010.

[GXD+14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pp. 599–613, 2014.

[KG11] Jeremy Kepner and John Gilbert, "Graph algorithms in the language of linear algebra," SIAM, 2011.

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

[KPW+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 211–222, 2007.

[LM11] Amy N Langville and Carl D Meyer, "Google's PageRank and beyond," Princeton university press, 2011.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 135–146, 2010.

[MAY23] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman, "Design and implementation for non-blocking execution in GraphBLAS: tradeoffs and performance," in *ACM Transactions on Architectures and Code Optimization* 20(1), Article 6:1-23, 2023.

[MAY22] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman, "Nonblocking execution in GraphBLAS," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 230–233, IEEE, 2022.

[MWM15] Robert Ryan McCune, Tim Weninger, and Greg Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

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

[PGZ+11] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P Johnson, and David I August, "Commutative set: A language extension for implicit parallel programming," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 1–11, 2011.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, IEEE, 2010.

[SM09] Alexander A Stepanov and Paul McJones, "Elements of programming," Addison-Wesley Professional, 2009.

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

[SV80] Yossi Shiloach and Uzi Vishkin, "An $\mathcal{O}(\log n)$ parallel connectivity algorithm," Technical report, Computer Science Department, Technion, 1980.

[SW14] Semih Salihoglu and Jennifer Widom, "Optimizing graph algorithms on Pregel-like systems," *Proceedings of the VLDB Endowment*, 7(7), 2014.

[SY19] Wijnand Suijlen and A. N. Yzelman, "Lightweight Parallel Foundations: a model-compliant communication layer," arXiv:1906.03196v1, 2019.

[Val90] Leslie G Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 33(8):103–111, 1990.

[YDNNS20] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen, "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation," 2020.

[ZAB20] Yongzhe Zhang, Ariful Azad, and Aydin Buluç, "Parallel algorithms for finding connected components using linear algebra," *Journal of Parallel and Distributed Computing*, 144:14–27, 2020.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.