HUMBLE HEROES

A. N. YZELMAN

Abstract. We first recall the concept of a *humble programmer*, coined by Dijkstra in the 1970s, which fully acknowledges the complexity of programming as being unsuitable for the human mind. This concept leads to humble frameworks that simplify not only the implementation of algorithms, but also automate the optimisation and scalability across compute units. Such humble programming models typically favour productivity over performance.

By contrast, *hero programmers* are experts with deep knowledge of parallel algorithms and related theory, coding, and hardware, typically seeking to achieve the highest possible efficiency on any given system. Since hardware complexity and diversity are set to increase in the years to come, the average programmer will no longer be able to drive new and potentially large-scale heterogeneous architectures, thus simultaneously increasing the pressure on hero programmers as well as highlighting the need for humble programming models.

This paper first sketches the successful humble MapReduce and Pregel vertex-centric frameworks, and identifies common factors in their design that are shared with the novel Algebraic Programming (ALP) paradigm. Balancing the need for supporting multiple humble programming models with reducing demands on hero programmers, the main contribution of this paper provides a vertex-centric programming model on top of ALP, thus demonstrating that multiple humble programming models may be supported by a single software stack, the latter of which serves as a natural focus area for the relatively few hero programmers.

Experiments demonstrate that the resulting ALP/Pregel interface scales on a shared-memory parallel system, achieving speedups of up to 17.8× on common graph workloads. Even though humble vertex-centric algorithms that solve a given problem commonly differ from their canonical solutions, the paper furthermore compares a vertex-centric algorithm for ranking web pages versus a highly-optimised canonical solution. ALP/Pregel achieves up to 8.99× speedup in the sequential case, outperforming the canonical solution in 12 out of 13 datasets tested. In the shared-memory parallel case, ALP/Pregel achieves between 0.276–10.0× speedup versus the optimised baseline, with both extreme slowdowns and speedups mostly due to the work space fitting in L3 cache for the optimised baseline but not for the vertex-centric programs. For the smallest and largest datasets where cache effects are less disparate, ALP/Pregel is faster in 3 out of 6 instances with speedups up to 5.73×. Thus humble solutions may still achieve hero performance, as well as contribute to solving the looming software productivity crisis.

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 that only a small percentage of highly skilled programmers can write efficient programs.

The increasing difficulty in writing software is due to the increasing complexity of computer architectures, coupled with a wild-growth of architectures that programs should cope with. Recent hardware trends include increasingly non-uniform memory access (NUMA) designs –with already four to eight NUMA domains on modern CPUs– as well as a decreasing trend of memory capacity per core, in addition to a decreasing memory bandwidth per core. These factors will significantly increase pressure on future software design for any given architecture. Heterogeneity may appear on a spectrum: from within a single chip, consisting of different types of processing units, to large-scale data centres that connect a diverse range of architectures, including classical CPU architectures and accelerators. Heterogeneity will realistically continue to appear within and between these extremes.

With a growth of architecture targets with increasing design complexity and increasing 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 extract acceptable performance on a given architecture. The required breadth of expertise sets this class of programmers apart from the majority– colloquially, we refer to the these as the *hero programmer*.

1.1. **Humble programming.** In contrast to such hero programmers, Dijkstra first coined the concept of a *humble programmer*. It entails, paraphrasing from his monologue [Dij72], *approaching programming while fully recognising the incredible complexity of the task, by preferring modest and elegant programming languages that respect the limitations of the human mind*. Perhaps implicit with humble programming comes an acceptance of a performance trade-off, favouring programming productivity over performance. This paper provides evidence that the two targets of performance and productivity need not be mutually exclusive.

Contemporary examples of humble programming models that also scale across large-scale computational resources include MapReduce [DG08], Pregel [MAB+10], and Spark [ZCF+10], which allow driving complex large-scale systems using simple, elegant, and easy-to-use abstractions that may be grasped by any programmer. These programming models have been extremely successful at providing parallel compute power to the humble programmer, allowing them to quickly deploy and scale up a variety of workloads.

1.2. **The software bottleneck.** The hero programmer, by contrast, strives to achieve as close as possible to peak performance 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, low-level threading interfaces, remote direct memory access, message passing, and so on. The resulting code requires significant time and effort to build and is non-trivial to maintain, while re-tuning the code to novel versions of the architecture originally targeted is equally non-trivial. Furthermore, supporting completely novel architectures requires a ground-up duplication of all aforementioned efforts.

1.3. **The challenge.** If the future is to be increasingly heterogeneous in nature, then its corresponding software stack must not rely 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 the humble programmer, deploy to different architectures, *and* achieve good performance– while avoiding the software bottleneck.

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

Section 4 notes that while ALP may be a significant leap forward for anyone comfortable expressing algorithms using algebraic annotations, some humble programmers may prefer using alternative paradigms. It submits that neither forcing a single paradigm on humble programmers nor supporting an unchecked set of potentially many humble paradigms solves the challenge of productive usage of future large-scale heterogeneous systems, and motives the alternative approach this paper takes: the automatic translation of one humble programming model into another.

Section 5 realises this ideal for Pregel and ALP, by introducing a C++ API for vertex-centric programming that compile-time translates into ALP, thus supporting multiple humble paradigms with a single humble software stack. Section 6 presents experimental results that show that the cost of simulating a Pregel-style interface on top of ALP is negligible, and performs well in comparison to direct programming in ALP/GraphBLAS– and, if some relaxations are permissible, even outperforms it. Section 7 concludes and provides an outlook to future work.

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 select group of parallel programming experts only: MapReduce and Pregel.

2.1. **MapReduce.** With MapReduce, data items are key-value pairs (k, v) from some domain K and V, respectively, while 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 $f_i(d_k)$; thus, the original pair d_k may be filtered out, generate exactly one new pair, or generate arbitrarily many new pairs. 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 old set of data D_{2i} is discarded and replaced by the set of transformed entries D_{2i+1}.

The i-th *reduce* phase operates over all keys in the current data set D_{2i+1}, i.e., over the set $K_i = \{k \in K \text{ s.t. } \exists (k, v) \in D_{2i+1}\}$. Assume there are, for one such key k, $r > 0$ pairs $(k, v_0), \ldots, (k, v_{r-1})$. Then the reduction computes $v' = \odot_{i=0}^{r-1} v_i$, and produces an entry (k, v') in $D_{2(i+1)}$ that replaces all pairs with key k in D_{2i+1}– this proceeds for each key in K_i. Like for the map operation, the reduction operator $\odot$ is user-defined and may differ for each round.

While the map operations may be arbitrary, it is helpful if the reduction operators are associative. If so, then the reduction can be automatically parallelised by systems like ALP. If reduction is not associative, then the reduction phase may only be parallelised embarrasingly over each key $k \in K_i$, thus limiting the amount of parallelism exposed.

The round-based nature of a MapReduce communication, especially that the map phase induces no communication between parallel compute units while the reduce phase does require communication, is closely related to the Bulk Synchronous Parallel (BSP) paradigm of parallel computing [Val90]. 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 which, like MapReduce, proceeds in rounds which each have two phases: first, local computations using local memory elements must complete; second, arbitrary data movement patterns between memory elements must complete. On any one compute unit, a next round may only proceed when all incoming messages and outgoing messages have been received, respectively, sent. This does not imply a global synchronisation barrier between rounds.

The differences between direct-mode BSP and MapReduce are two: 1) BSP operates on the granularity of the distributed machine itself, while MapReduce operates on the granularity of the dataset sizes $|D_i|$; 2) BSP allows realising arbitrary communication patterns, while MapReduce always sorts on keys. Valiant, however, also proposes an *automatic mode* of BSP that allows the simulation of PRAM algorithms on a BSP architecture. The hashing mechanism by which this proceeds is typically mirrored in MapReduce implementations such as Hadoop [SKRC10], as well as in the open-source Pregel realisation Giraph [Ave11] and the Spark framework [ZCF+10].

Such simulation techniques are not free, and precludes the application of communication-optimal algorithms. Thus, orders of magnitude performance differences between humble programming frameworks and the direct-mode BSP are not uncommon; see, e.g., Suijlen and Yzelman for a performance comparison between ALP/GraphBLAS and Spark [SY19]. They report a two-orders of magnitude performance difference for PageRank on moderate data sizes (approx. 936 million edges) using ten nodes.

2.2. **Pregel.** Pregel is a programming model for graph computations [MAB+10]. Suppose the data graph $G = (V, E)$, then Pregel allows defining a round-based computation that each vertex $v \in V$ executes. Each round consists of two steps: 1) the execution of the user-defined program on each vertex; and 2) the exchange of messages via the edges E. Originally, and without loss of generality, messages take place as broadcasts from vertices $v \in V$, sending some message $m \in M$ to all neighbours $N(v) = \{w \in V \mid (v, w) \in E\}$ of v.

This type of *vertex-centric* programming model has been very successful and has inspired a significant number of vertex- and edge-centric programming frameworks [MWM15]. Its round-based nature, like that of MapReduce, is again a variation of the Bulk Synchronous Parallel (BSP) model, again applied in a fine-grained fashion by mapping the programs on the input data itself. Most variants of Pregel have come into existence by realising that while the algorithm model is strictly round-based, the execution of such a program need not be; 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 (SCC) of a graph $G = (V, E)$, which seeks to identify the minimum number of subsets $V_k \in V$ for which $(V_i, E), \forall i$, are strongly connected[1]. A simple vertex-centric program determines these subsets by first assigning every $v \in V$ a unique identifier. Each vertex then, during each round, 1) sends this current identifier to its neighbouring vertices, and 2) overwrites its current identifier to the maximum of all incoming messages, if strictly larger than the previous identifier. If during the latter stage the previous identifier is not overwritten, the vertex furthermore votes to halt the program. Execution terminates when all vertex programs vote to halt Once terminated, the number of strongly connected components k corresponds to the unique identifiers that remain while the component each vertex belongs to is indicated by its identifier.

This vertex-centric algorithm and its stopping criterion are intuitive to understand, hence fitting the bill of a humble solution to solving the SCC problem. However, considering an (undirected) line graph as an example input to this algorithm, the number of rounds it takes will be $|V|$. Furthermore, $|V|$ vertex programs are active per round, thus amounting to $\Omega(|V|^2)$ work for this SCC algorithm when executed on a line graph. By contrast, the Awerbuch-Shiloach [SV80, AS87] algorithm for SCC can be practically realised to achieve $\mathcal{O}(|E| \log |V|)$ work complexity. Algorithms with such improved bounds may 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 given e.g. 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 regardless, in which case the humble choice suffices.

This paper, in line with its motivation, considers only easy-to-understand humble algorithms, leaving the important task of finding and realising asymptotically optimal algorithms in any framework, to the hero programmer.

2.3. **Common factors.** Finally, we remark on shared concepts between the successful MapReduce and Pregel programming models. First, the frameworks operate on sets of data, and express parallelism chiefly by operating concurrently on data elements within those sets. This is also known as a *data-centric* approach. Second, the expression of parallelism is implicit: operations are expressed in terms of mutating one set of data into another. Third, operations proceed round-by-round, thus exposing a sequential view of the final programs, much akin to the direct-mode of BSP and traditional imperative programming. Fourth, efficient implementations of these humble programming models exploit the imposed program structure to enable both scalable execution as well as a high degree of overlap between executing rounds and phases, by overlapping communication with computation as well as by exploiting parallel slackness [Val90, MAB+10]. Additionally, these four properties are shared with another successful humble programming model, Spark [ZCF+10].

3. Algebraic Programming

The ALP paradigm provides a data-centric, sequential, and imperative programming approach based on algebraic concepts and annotations. It is designed to be humble, and exposes three algebraic concepts to programmers: containers, structures, and primitives. ALP/GraphBLAS specifically, introduced by Yzelman et al. as a C++ realisation of the GraphBLAS [YDNNS20, BMM+17, BBM+19], focuses on sparse linear algebra. Containers there take the form of vectors and matrices; structures may be binary operators, monoids, or semirings; while operations may be the element-wise application of a binary operator, a matrix–matrix multiplication, and so on.

3.1. **Algebraic containers.** Containers in ALP, on declaration, will initially be *empty*– meaning they contain no values. The minimum capacity of a container will by default equal its maximum dimension. Consequently, a vector of size n after creation will hold no values and have a capacity for n values, while a novel $m \times n$ matrix will similarly hold no values while its default capacity shall be at least $\max\{m, n\}$. Larger (or smaller) capacities may be passed through container

[1] A (sub)graph is strongly connected if and only if there exists a path between any two vertices in that (sub)graph.

creation, while existing containers may be resized to potentially hold a larger number of values through grb::resize. An ALP implementation may elect to assign capacities that are larger than requested, but may not assign smaller-than-requested capacities; if a requested minimum capacity cannot be guaranteed, an error is returned instead.

Using iterators to STL-compatible C++ containers, data may be *ingested* into ALP containers using grb::buildVectorUnique and grb::buildMatrixUnique. The '-Unique' postfix in the aforementioned two primitives indicates that the source container shall not contain duplicate entries, meaning that there shall not be multiple values that map to the same container coordinate. The current size of a vector is returned via grb::size, the row and column size of a matrix is returned via grb::nrows and grb::ncols, respectively, while the current number of values in a container may be referenced via grb::nnz. All values may be erased via grb::clear.

Going by example, the following three statements create a vector with default capacity, a vector with potentially smaller capacity, and a matrix with an exact initial capacity as returned by a matrix file parser:

```
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 a container appears as a template argument, which conforms also to the C++ Standard Template Library (STL).

Extraction of data from containers proceeds using iterators that can be retrieved through the begin, cbegin, end, and cend functions. These output iterators likewise conform to the STL, but only support **const**−variants: values in containers may not be adapted through iterators. Aside from data ingestion through iterators, programmers may also set a vector to 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 );
```

After all above example code have executed x corresponds to a dense vector $(1, 1, \ldots, 1)$, while y corresponds to a sparse vector where only one nonzero y_i exists with value (*true*).

3.2. **Algebraic structures and algebraic type traits.** An example of a binary operator is addition over double-precision floating point numbers, which ALP exposes as a C++ template class:

```
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 $(a \odot b = b \odot a)$, or idempotency $(a \odot a = a)$. ALP/GraphBLAS exposes such properties through *algebraic type traits*, such as, 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 may be inspected at compile-time, thus enabling semantic checks and compile-time optimisations guided by algebraic properties.

Richer algebraic structures include monoids and semirings, which may be *composed* from operators and identities. For example, the following type

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

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

For any binary operator $\odot : D_1 \times D_2 \to D_3$, the three domains potentially differ. Consider, for example,

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

which describes the argmax operator over tuples of integers and floating point numbers that results 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 an "infinity" over a tuple is composed by recursion over the tuple types– in this case, resolving 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, the expression

```
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 given, here simple numerical addition. Hence y, taking in mind the earlier examples that defined and populated x, after executing the above reads $y = (2, 2, \ldots, 2)$.

Consider element-wise application in the sparse case:

```
grb::Vector< bool > 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. Since 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 those 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. The above thus results in a single entry to d, 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 contrasts the behaviours of numerical multiplication as an operator versus as an monoid, under element-wise application:

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

```
grb::eWiseApply( y, x, oneTwo, mulOp );   // y is equal to 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 operate in an out-of-place fashion, while all other primitives have in-place semantics; i.e., the above grb::mxv sets y to $y \oplus Ax$, with $\oplus$ the additive operator of the given semiring, and not to Ax. If the latter is intended the output container must be cleared first, as in the above.

All primitives with container output support masking. For example,

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

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

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

3.4. **Element-wise lambdas.** Yzelman et al. recognised that in a *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. Consider, for example,

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

Should the two primitives be executed one-by-one, then elements from x and y are brought from main memory to the CPU core(s) twice[2], while once would have sufficed if the two calls were fused instead. Hence Yzelman et al. introduced the grb::eWiseLambda that allows the execution of arbitrary lambda functions on one or more vector containers. The above snippet, for example, could be replaced with the semantically equivalent

```
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 accesses 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) used inside a lambda function passed to grb::eWiseLambda, and 2) the index i refers to a nonzero value that already existed in the related container when the eWiseLambda was invoked. Any other use

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

invites undefined behaviour. The first vector argument to grb::eWiseLambda (second-to-last line, x) defines which indices the eWiseLambda shall iterate over, while the other vector arguments (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 nonzeroes of x unmodified:

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

This element-wise lambda, outside its intended use case of manually fusing ALP operations, provides critical functionality for translating vertex-centric programs into ALP as Section 5.4 shows. More in the context of humble programming, recent work by Mastoras et al. provide mechanisms by which ALP/GraphBLAS may automatically fuse operations [MAY22a, MAY22b]– Section 7.2 discusses the implication of this recent development for vertex-centric programs.

3.5. **Final remarks.** All ALP primitives return error codes that this paper shall ignore in favour of brevity. A special note, however, is that the program must guarantee sufficient capacities in output containers or otherwise the related operation may fail. For example,

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

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

While this section has not introduced the complete ALP/GraphBLAS API nor that of other ALP extensions, the material covered provides sufficient basis for the remainder of this paper. For full details, please see the respective papers [BMM+17, YDNNS20, MAY22a, MAY22b].

4. Motivation

The ALP paradigm expresses programs as a sequential, data-centric, and standard C++ programs, and takes care of performance aspects and corresponding code optimisations and parallelisation, fully automatically. It enables, for example, distributed-memory parallel computations that 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 that the ALP programmer annotates their program with: ALP makes use of exactly those high-level annotations to select the appropriate optimisations automatically, while the programmer can focus solely on the mathematics [YDNNS20].

However, while most programmers at some point during their studies have studied linear algebra, most programmers likewise do not make daily use of linear algebraic concepts– including, in particular, the conscious use of monoids or semirings. This limits the humble appeal of ALP. Separately, another key question asks how many workloads would naturally map to the ALP paradigm.

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

(1) educate programmers in the use of algebraic concepts in programming;
(2) extend ALP functionalities to support broader ranges of workloads;
(3) expand other humble programming models into ALP.

This paper primarily focuses on the third approach, though it contributes to the former former two also.

Educate. No programming model or language has been successful without educating programmers in their use. In the case of ALP/GraphBLAS this may have limited complexity as most programmers already make implicit use of algebraic concepts– whether this be linear algebra and its implicit use of the real semiring, or set algebraic concepts like orders implicitly used when

sorting. Furthermore, the idea that programming should follow more closely mathematical concepts is not new; it forms the basis behind the design of the C++ Standard Template Library 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 of e.g. the explicit use of semirings in programming including the seminal works by Aho, Hopcroft, and Ullman [AHU74] and Cormen, Leiserson and Rivest [CLR92].

Extend. Separately, ongoing research should push the boundaries of ALP applicability to workloads 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 suggests two other extensions that would enable the automatic translation of MapReduce programs into ALP. Similar extensions are set to enlarge the scope of ALP applicability and is subject of ongoing research. Important is that such extensions remain minimal as well as faithful: 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.

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 mandates that programmers should always be allowed the use of the tools that they consider most intuitively suitable for different programming tasks. However, unchecked growth in software stacks supporting different programming paradigms, many architectures, and many heterogeneous configurations, does not scale– the number of programming tools with distinct software stacks should be minimised.

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. In this view, many humble programming models could exist, though only very few should be fundamental. The many humble models should automatically translate to a fundamental one, while only fundamental programming models are backed by an automatically optimising, parallelising 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 sharing the same software stack, demonstrates both the performance and scalability of the approach, and as such motives ALP as a fundamental humble programming model. With this solution, humble programmers may rely on multiple programming interfaces and select the most suitable ones for each job, while hero programmers may 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. The interface describes a pure vertex-centric interface without using ALP concepts, save for a monoid over which incoming messages are to be reduced. Using this interface, the SCC algorithm is revisited and precisely formulated, while additionally introducing an ALP/Pregel program for web page ranking based on the PageRank algorithm.

5.1. **Interface.**

The C++ interface supports two mechanisms for determining termination conditions: vote-to-halt, and inactivation of vertex programs. In the former, all vertex programs vote on whether to terminate; only if all active vertices vote to halt, will the program indeed terminate. With the latter mechanism, 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.

The interface is most concisely introduced by example. Algorithm 1 shows the strongly connected components algorithm in our vertex-centric API, while Algorithm 2 shows a simplified PageRank algorithm. Both algorithms are located in the grb::algorithms::pregel name space.

Algorithm 1 Vertex-centric strongly connected components

```
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;
                }
        }

};
```

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

5.2. **Strongly connected components.** Briefly recapitulating the earlier SCC example, every vertex is initially assigned a unique identifier (ID) integer. Then each vertex broadcasts their ID, overwriting their local ID by the maximum ID received. If this leads to no update (the current ID is already the maximum), then the program votes to halt execution. In-line with being a humble programming model, the reader may well intuitively find this algorithm indeed converges to a correct solution where every component will a unique ID that corresponds to the maximum initially assigned of any vertex in the component.

The ID type may be any integer such as **unsigned int** or **size_t**, while incoming and outgoing messages are of that 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) and out-degree (line 10) meta-data that our Pregel API provides, and makes use of the

Algorithm 2 Vertex-centric PageRank

```
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);
                }
        }
};
```

vote-to-halt mechanism (line 13). 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 initialisation of the Pregel interface over a specific graph. Whereas normally in ALP/GraphBLAS a graph file is opened using a parser and then ingested into a grb::Matrix instance, such as, e.g., 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 are to be ignored, as indeed Pregel algorithms employ the edge structures for determining the message patterns. The template argument nonetheless remains for future extensions that may consider edge-centric, or a combination of vertex- and edge-centric, programming paradigms.

Once the Pregel instance is instantiated, it may execute any Pregel algorithm on the underlying graph. This employs the execute member function of the Pregel instance; e.g., 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 where the 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 equal to the same integer type. Using a monoid structure rather than an arbitrary message combiner function helps ensure, for a specific vertex, that if

(1) no incoming messages are received, for example because a vertex in-degree is zero or because all vertices with edges incident to this vertex have become inactive, then the monoid identity can be substituted as a received message and thus ensure consistent behaviour;
(2) multiple messages are received in orders that potentially differ across rounds and executions, then the commutativity of the combiner function ensures consistent behaviour.

Since in the strongly connected components example we select the maximum component ID, 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 not supplied, the program will run until a termination condition arises.

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

5.3. **Vertex-centric PageRank.** 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], yet is a common approximation that appears in vertex-centric and Spark-based literature; see, e.g., Gonzalez et al. [GXD+14]. The algorithm has two canonical algorithm parameters: α and *tol*. For the former, α may be viewed as the regularisation parameter or, more intuitively, as the probability a surfer on the web visits a random other web page rather than a random link on a current page. The latter parameter, *tol*, signifies a desired local tolerance that, once met, will set the current vertex to inactive (line 17 of the vertex-centric program).

The local state of every vertex is its PageRank score, here represented by some type IOType (e.g., **double**). Each round of the vertex-centric program distributes the local score equally to all neighbours of the current node; that is, the current score s is divided across all neighbouring 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 old score is less than *tol*, the program considers the local vertex converged and removes itself from further rounds of computation, while for its active neighbouring nodes it shall seem as though 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 PregelState::round field which keeps track of the current round of computation, and employs it to initialise the vertex weights during the first round (lines 1–7). In initialisation, it furthermore demonstrates how global information on the total number of vertices in the program (PregelState::num_vertices) may be employed within vertex-centric programs by normalising the initial PageRank score[3]. Line 17 shows how a vertex removes itself from computation, while the mechanisms by which the outgoing message is broadcast (lines 21–25) introduce no further novel concepts.

This program is executed on a Pregel instance in much the same way as with the previous example, except that vertex weights and messages are now of type **double** instead of **size_t**, and except that a regular additive monoid is used instead of a max monoid. As a convention, however, 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::RC rc = grb::set( pr_scores, 0 );
rc = rc ? rc : grb::algorithms::pregel::PageRank< double >::execute(
    pregel, pr_scores, rounds_taken
);
```

Termination of this program is not theoretically guaranteed especially when limiting the number of rounds to a small number or when choosing a low α [LM11]; Section 6 takes care to report only experiments where execution the program has converged.

As noted earlier, this PageRank algorithm does not correspond to the canonical version by Brin and Page: it misses contributions by dangling nodes[4]. A common extension to the original Pregel framework [MAB+10] adds global aggregation mechanisms. These would allow for the correct implementation of the PageRank algorithm by correcting the PageRank updates with contributions by dangling nodes, as well as allow for global convergence detection, and is provided e.g. by the Giraph [Ave11] vertex-centric framework. ALP/Pregel does not provide these as the author believe it limits automatic optimisation opportunities that would overlap the execution of rounds, discussed in more detail in Section 7.2.

In partial as an alternative, ALP/Pregel instead supports removing vertices from execution permanently by setting them inactive– while it is a common optimisation for improving convergence

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

[4]Nodes with out-degree zero.

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 of the computation
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

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

for e.g. the PageRank algorithm [LM11], it is not supported by all vertex-centric frameworks. While this mechanism may accelerate convergence for this particular algorithm, ALP/Pregel also exploits inactive vertices to accelerate the per-round computation speed, as Section 5.4 details. The examples thus far have introduced all fields available in grb::interfaces::PregelState except for num_edges. Table 1 summarises all available fields.

5.4. **Implementation.** The implementation of the vertex-centric ALP/Pregel interface compile-time translates to a while-loop around standard ALP/GraphBLAS primitives. Each iterand of the while-loop corresponds to the execution of one round of computation and the subsequent termination detection and message exchange. Prior to the first round, all vertices are added to the *active list* while the outgoing message buffer 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;
(8) exchange messages and aggregate incoming messages via grb::vxm.

The latter 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 nonzeroes 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 initialisation of a grb::interfaces::Pregel instance, the active list is initialised to **true** for all vertices, while the in- and out-degrees are computed using SpMV multiplication of the graph adjacency matrix with a vector of ones. The vertexIDs 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 incoming message vector is user-defined and its values may be updated by the vertex-centric program, while the outgoing messages are reset each round via

```
grb::set< grb::descriptors::structural >(
```

```
    outgoingMessages, activeList, id
);
```

Here, id is the identity of the aggregation monoid of the same type as that of outgoing messages. Note that this updates only the outgoingMessages of active vertices.

Calling the user program. The user-program executes through a call to grb::eWiseLambda, which first captures, for every active vertex, a reference to the corresponding vector entries of the active list, in- and out-going messages, state, in- and out-degrees, and vertex IDs. It then calls the user-defined program. This 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 nonzeroes 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 a previously active node to false, which does require actual Boolean values be present in the activeList. 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 is the only step primitive in the ALP/Pregel implementation without that descriptor. The use of the masked set operation ensures that the number of nonzeroes in the buffer has a reduced number of nonzeroes. Finally, the buffer is swapped with the active list to be used as the new mask for subsequent steps of the algorithm.

The implementation maintains an extra buffer to support this update step. According to the performance semantics ALP/GraphBLAS defines, the grb::set on a non-empty output implies a cost proportional to the old number of nonzeroes plus, perhaps more obviously, a cost proportional to the new number of nonzeroes [YDNNS20]. Thus while the buffer takes up $\Theta(n)$ memory, the overhead of this update step bounded as $\Theta(k)$, with k the old number of active vertices.

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

Message exchange. That we may perform the message exchange and aggregation using vector–matrix multiplication may be unclear without describing the semiring under which this proceeds. The given aggregation monoid plays the role of 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 $x \otimes y = \mathit{left\_assign\_if}(x, y)$, with x from a user-defined domain D and $y \in \{\mathit{false}, \mathit{true}\}$, then $\mathit{left\_assign\_if} : D \times \{\mathit{false}, \mathit{true}\} \to D$ is defined as follows:

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

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

The thus composed semiring is, in fact, an improper one: the aggregator identity, since it may be user-defined, may not necessarily annihilate under the multiplicative operator. If the additive monoid is the logical-or with **false** as the additive identity, however, the thus composed semiring with left_assign_if is, in fact, proper, since it reduces to the standard Boolean semiring.

The way we use the semiring guarantees that the multiplicative operator is only ever called using the multiplicative identity as the right-hand side input argument, thus ensuring the improper cases

for non-logical-or aggregators do not trigger. This careful usage proceeds as follows. The grb::vxm operation $in = outG$ matches nonzeroes $g_{ij} \in G$ to corresponding elements out_i from the outgoing message buffer, and first applies the multiplicative operator to form a temporary $tmp = out_i \otimes G_{ij}$. Since 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. In other words, the ALP/Pregel implementation ensures that multiplication with zero never occurs.

5.5. **Summary and costing.** The described implementation of vertex-centric programming in ALP/GraphBLAS makes ample use of its main features:

(1) composability of monoids and semirings provide the flexibility to integrate the user-defined aggregation into the richer algebraic semiring structure;
(2) grb::eWiseLambda which allows the execution of any user-defined operation on vertex (vector) data;
(3) exploiting sparsity, masks, and algebraic structures 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. These correspond to a $\Theta(n)$ memory usage for executing any Pregel program. Execution does not allocate nor free any other memory while operating. Computing termination conditions consist of $\Theta(k)$ work, with k the number of active vertices in a given round. Capturing vector elements as arguments to user-defined vertex-centric programs has $\mathcal{O}(1)$ overhead in the worst case, translating to $\mathcal{O}(k)$ costs per round. The cost of executing a vector program is determined by the user, and is linear in k– i.e., for a vertex-centric program that takes $\Theta(1)$ time, the total cost of executing all active programs is $\Theta(k)$.

The worst-case data movement complexities in a shared-memory setting correspond to $\Theta(k)$ with regards 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, then again the total memory movement is $\Theta(k)$ per round. Assuming that similarly the sizes of incoming and outgoing messages are $\Theta(1)$, then the message exchange takes

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

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

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

For distributed-memory parallelisation, again work may be divided by P while adding a factor $\mathcal{O}(P)$ to all storage, work, and data movement bounds, where P is the number of distributed-memory processes. Additionally, the active list update and the vote-to-halt termination check amount to a reduction across all nodes, which induces $\mathcal{O}(P)$ inter-process data movement and work, as well as up to $\mathcal{O}(\log P)$ synchronisation 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 summarise 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 as well as intra- and inter-process data movement improve.

	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 2. Costs of executing a single round of an ALP/Pregel program on a shared-memory machine. Assumes n vertices of which k 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, while 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 $T = 1$.

	Work	Data movement	Synchronisations
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	$\Theta(k)+\mathcal{O}(\log P)$	$\Theta(k)+\mathcal{O}(\log P)$	$\mathcal{O}(\log P)$

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 that of Table 2, wherein T in division should be replaced with the number of threads available at each process multiplied with the number of distributed-memory processes P. Additionally, all its data movement costings should be divided by P, while all three of storage, work, and data movement add a factor $\mathcal{O}(P)$.

6. EXPERIMENTS

Experiments are constructed to 1) show that humble ALP/Pregel programs achieve the scalability predicted in the previous section, and 2) to show that ALP/Pregel is competitive with state-of-the-art frameworks. For the former, the strongly connected components (SCC) algorithm is investigated, comparing both sequential and parallel results. Then, using two variants of the PageRank algorithm, the behaviour of ALP/Pregel under increasingly many inactive vertices is investigated. Finally, a scalability experiment using PageRank algorithms demonstrates that shared-memory auto-parallelisation of ALP/Pregel behaves as expected also when inactive vertices increase.

For a comparison against the state-of-the-art, for ALP/GraphBLAS performance, we first summarise from recent work by Mastoras et al. [MAY22a] who provide an in-depth comparison 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× faster than a PageRank implemented in SuiteSparse:GraphBLAS, while being 0.64–12.2× faster than one written using Eigen on shared-memory parallel architectures. Both SuiteSparse and Eigen auto-parallelise over shared-memory architectures, while GSL does not. Eigen additionally performs loop fusion which leads to speedups versus ALP/GraphBLAS for small matrices. Mastoras et al. obtain similar results are obtained for two other sparse matrix and graph algorithms.

Rather than duplicating work, this paper instead focuses on comparing the ALP/Pregel PageRank-like algorithm while taking the canonical ALP/GraphBLAS variant as a state-of-the-art baseline. It furthermore does not compare against existing vertex-centric frameworks such as Giraph [Ave11], since such frameworks rely on file-based fault tolerance will not compare well versus the state of the art, typically resulting in orders-of-magnitude performance losses versus optimised optimised

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

TABLE 4. The graphs used in experiments. All except for 11 and 12 are undirected.

code [SY19]. Since we look for humble programming models that achieve 'hero-level' performance, such comparisons are out of scope.

6.1. **Methodology.** Experiments are run on a dual-socket Intel x86 machine, consisting of two Intel Xeon 6238T processors, each of which have 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. Cores are clocked at 1.9 GHz, have hyperthreads enabled, and turbo boost disabled. Each processor has six memory channels clocked at 2 933 MT/s, for 262.2 GB/s 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 since ALP/Pregel computations are typically memory-bound, obtaining much less speedup than the total number of cores would indicate. For this architecture, hence, a bandwidth-bound application should see between 14.3–26.4× speedup.

Our implementation and results are based on v0.6 of ALP/GraphBLAS as available on GitHub and Gitee [Alg21a, Alg21b]. Its sequential reference and shared-memory parallel reference_omp backends are used to both compile ALP/Pregel programs with, as well as to provide a baseline PageRank implementation. All software is compiled using GCC 9.3.1 using the `-O3 -mtune=native -march=native -DNDEBUG -funroll-loops` compiler flags. All compilations and experiments execute under a Linux distribution with 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 so to that one timing takes at least 100 milliseconds. Then, experiments are repeated at least 10 times in order to compute a sample standard deviation across all timings. All reported timings have a sample standard deviation of less than 3 percent of the measured average time. Timings with sample standard deviation higher than 1 of the average time are printed in *italics*, while the tables only report obtained averages at three significant digits; these thus are accurate to a significant degree for timings printed in non-italics, while the least significant digit is likely incorrect for timings printed in italics.

Exceptions to this methodology include the sequential ALP/Pregel SCC, for which due to a very long run-time, only three experiments are performed for most datasets.

6.2. **Datasets.** Our experiments require input graphs to run the vertex-centric algorithms on. Typical datasets that vertex-centric frameworks operate on include knowledge graphs, graph representations of the world wide web, and other types of networks. To also compare against wildly different structures than such typical graphs and so gauge the effectiveness of the paradigm when faced with problems originating from other domains, our dataset in Table 4 includes both graphs and sparse matrices from various domains that appear in scientific computing.

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×

TABLE 5. The run-time in milliseconds of executing the ALP/Pregel SCC, Algorithm 1, compiled using the sequential and shared-memory parallel ALP/GraphBLAS backends. The number of iterations follow each timing in parenthesis, while the last column reports the speedup of the parallel execution versus the sequential one.

All datasets originate from the SuiteSparse MatrixMarket collection [DH11], and are ordered by the number of vertices they encode. On the architecture described earlier, 128k 64-bit words (**double**s or **size_t**s) fit into private L2 caches, and hence we expect significant reuse for 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 from L3 should occur for datasets 3–10; the larger datasets will induce out-of-cache behaviour always. As will be elucidated later in this section, performance differences between some PageRank implementations relate to the number of edges per vertex, which Table 4 hence reports as well.

Aside from differences across the number of vertices and edges, we augment typical graphs which have no discernible structure when plotting the adjacency matrix with structured matrices. Such structures include having a fixed number of diagonals in the adjacency matrix, or only nonzeroes within a fixed bandwidth around the main diagonal. Such datasets induce favourable data access patterns that modern hardware prefetchers implicitly exploit, thus inducing very different behaviour during computations on such graphs. These effects are well-known in high-performance computing research dealing with sparse matrices. Some datasets such as G3_Circuit have both structured and unstructured components in the adjacency matrix, while another, bundle_adj, has an adversarial structure that can have major impact on the performance of linear algebraic libraries and programming frameworks [MAY22a]. All graphs the adjacency matrices represent are undirected, except for wikipedia-20070206 and uk-2002.

6.3. **Strongly connected components.** We first consider the scalability of ALP/Pregel using the SCC in Algorithm 1. Recall that for this algorithm all vertices remain active in every round. Table 5 compares the wall-clock time of executing the related ALP/Pregel SCC algorithm, once using the sequential ALP/GraphBLAS backend, and a second time using the shared-memory parallel OpenMP-enabled backend. We emphasise that the parallelisation is fully automatic and requires no change from the code presented in Algorithm 1.

The number of rounds required by the algorithm is reported in parenthesis, while the table also reports speedup obtained by shared-memory parallelisation.

The SCC algorithm employs solely the vote-to-halt mechanism, so each round executes with all vertices active. Furthermore, parallelisation should not affect the number of rounds the computation requires. This experiment hence measures purely the scalability of the shared-memory parallel backend.

Parallelisation comes at unavoidable overheads of at least $\Omega(\log T)$, in ALP/GraphBLAS bounded by $\mathcal{O}(T)$ [YDNNS20]. Therefore certainly for smaller datasets on the full system with 88 hyper-threads and two NUMA domains, speedups are expected to lie far below the 26.4× speedup for the triad benchmarks[5]. Since furthermore the vote-to-halt mechanism depends on a vector-to-scalar reduction, a significant portion of the vertex-centric paradigm will be bound by the 14.3× speedup for the reduction benchmark[5]. However, as datasets increase, speedups are expected to reach the 14.3–26.4× range.

As expected, Table 5 reports the same number of rounds for both the sequential and shared-memory parallel ALP/Pregel SCC algorithm. Its maximum speedup, 17.5× for dataset 11, indeed falls within the above range, indicating that parallelisation of the non-reduction parts of the ALP/-Pregel program achieves speedups closer to the best-case triad benchmark. Thus the ALP/Pregel implementation scales on shared-memory architectures, for the specific case when all vertex remain active throughout the computation.

6.4. **Sequential PageRank.** The PageRank from Algorithm 2, by contrast, does make use of 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 both the effect of the vertex inactivation mechanism on the speed of computation, we introduce a 'global' variant of the ALP/Pregel PageRank that modifies Algorithm 2 to, on local convergence detection, use the vote-to-halt mechanism instead of the inactivation mechanism. The original Algorithm 2 henceforth is referred to as the 'local' variant, as it does not consider a global inf-norm computation but rather a greedy local version of it.

Both the global and local ALP/Pregel PageRank algorithms are furthermore compared versus 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 cannot directly compare errors between the ALP/Pregel implementations and the ALP/GraphBLAS implementation: the algorithms are different and converge to different values. Recall from Section 5.3 that nonetheless the vertex-centric PageRank sees common application, most commonly in its global variant. Table 6 hence compares the local vertex-centric variant versus its global variant in terms of performance and speed of convergence, as well as compares both Pregel variants and the canonical PageRank approaches to web page ranking in terms of performance.

In terms of number of iterations, since we are comparing three different algorithms, it is *not* the case that the vertex-centric algorithms should always incur fewer iterations than the canonical PageRank, nor that the local variant of the ALP/Pregel PageRank should always incur fewer iterations than its global variant– nor would other such statement a priori be accurate. Indeed Table 6 shows that any one algorithm can incur the fewest number of iterations, depending on the input graph. The vertex-centric variants do require significantly more iterations for the large undirected graphs, IDs 11 and 12, to converge; this may well be due to common presence of dangling nodes in those graphs.

In terms of execution speed, the local variant should result in faster execution than the global variant, since as rounds progress, increasingly many vertices become inactive. Indeed the time per iteration reported in Table 6 confirms this, from whence we may conclude that our ALP/Pregel implementation indeed becomes faster as the number of active nodes decreases. One disadvantage of the use of the element-wise lambda noted by Yzelman et al. corresponds to the compiler no longer being able to apply vectorisation 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]Please see the ALP/GraphBLAS repository [Alg21a] for the full algorithm.

[7]I.e., dangling node corrections and 2−norm computations.

	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**

TABLE 6. Results, in milliseconds, for executing sequential PageRank algorithms, comparing the ALP/Pregel variants using global and local convergence versus the baseline ALP/GraphBLAS PageRank implementation. The number of iterations until convergence is reported within parentheses. The last three columns report the time per iteration, with the fastest in **bold**.

The time per iteration reported in the Table indeed shows that the ALP/GraphBLAS baseline dominates the global variant– while even outperforming the local vertex-centric variant in six out of thirteen cases. For the larger graphs this is explained by the observation that low edge per vertex count favour vectorisation (datasets 6, 9, and 13). Additionally a performance benefit for the pure ALP/GraphBLAS algorithm on small graphs is observed; this effect magnifies when parallelised, as presented and analysed in more detail in the next section.

In summary, in time per iteration, the local ALP/Pregel PageRank is up to 16.8× faster than its global variant and up to 6.84× faster than a state-of-the-art canonical PageRank implementation. In end-to-end run-time, the local variant is up to 7.48× faster than the global one and up to 8.99× faster than the state of the art baseline, while the biggest slowdown is limited at 0.95×. The local ALP/Pregel algorithm is fastest overall in twelve out of thirteen cases.

6.5. **Shared-memory parallel PageRank.** Similar to the SCC algorithm, parallelisation of the PageRank should not affect the number of iterations when assuming exact arithmetic. Numerical error propagation could cause minor variations, though comparing Tables 6 and 7 reveals no difference in the required number of iterations for our algorithms and datasets.

In terms of execution speed, there remains no instance where the global variant outperforms the local variant in end-to-end runtime nor time per iteration. In end-to-end run-time, the state-of-the-art PageRank implemented directly on top of ALP/GraphBLAS is fastest in eight cases; these instances also match the eight cases where the canonical PageRank has the fastest time per iteration. This is a departure from the sequential case, where the local variant had the fastest end-to-end run-time in all but one case, even though in time per iteration the baseline often outperformed the local variant. Indeed, differences in time per iteration have become more pronounced between the state-of-the-art and the local variant, with speedups now ranging from 0.403 to 10×.

Table 8 collects the speedups for all variants, similar to the SCC study in Section 6.3. The significantly higher ALP/Pregel speedups on bundle_adj (9.24–11.0×) compared to the baseline (1.81×) stands out. This is due to the adversarial structure of that graph, 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 a static schedule during SpMV multiplication. The ALP/Pregel variants, however, use a sparse structural mask for which the

Algorithm 3 A simplified representation of the ALP/GraphBLAS PageRank implementation[6]. Unlike Algorithm 2, this does faithfully implement the canonical PageRank [PBMW99, LM11].

```
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 );
}
```

backend reverts to a dynamic schedule during SpMV multiplication– which, in turn, avoids the imbalance induced by the adversarial graph structure.

The speedups for the ALP/GraphBLAS PageRank are generally better than those for the ALP/-Pregel variants, and the speedups for the global variant tend to be better than those for the local variant. While all implementations finally rely on the same OpenMP-enabled ALP/GraphBLAS backend, they rely on different OpenMP schedules: as discussed earlier, the ALP/GraphBLAS variant employs static scheduling during sparse matrix–vector multiplication, while 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 since the work load of the user-defined lambda may induce varying workload depending on the index it operates on. Additionally, some dense vector–vector operations in the baseline revert to a static schedule, while the same operations in the ALP/Pregel variants are part of the dynamically scheduled

	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 7. Results, in milliseconds, for executing shared-memory parallel PageRank algorithms. As in Table 7, compares the ALP/Pregel variants versus the baseline shared-memory parallel ALP/GraphBLAS variant.

	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

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

element-wise lambda. Some ALP/Pregel meta-data updates performed via level-1 operations in the global variant also employ a static schedule, while in the local variants reverting to a dynamic schedule.

Another major effect regards the work space of the ALP/Pregel programs. Recall that executing any Pregel program requires eight vectors, while the ALP/GraphBLAS PageRank implementation in Algorithm 3 requires half that. 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 materialisation of the messages between vertices in the form of incoming and outgoing messages, which means that computed values are copied twice more than in the baseline implementation which holds zero such copies.

The switch from static to dynamic schedules, the loss of data locality result, as well as the explicit materialisation of messages, result in a worst-case slow down of 0.340×, comparing the baseline and global variants. Additional uses of dynamic instead of static schedules in vector–vector and vector–scalar operations compound the worst-case slowdown to 0.212×, comparing

the baseline and local variants. The effects of reduced data locality should be reduced for larger problems.

While 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. The overheads due to message copies and loss of data reuse opportunities for small problems, however, remain unavoidable. Nevertheless, the local variant leads to the fastest end-to-end shared-memory parallel execution in five out of thirteen cases, with speedups up to 5.76× versus the baseline.

7. Conclusions and Outlook

The single foremost challenge in contemporary programming model and compiler research is to balance the need for easy-to-use programming interfaces that make future highly-parallel and highly-heterogeneous compute systems accessible to the humble programmer, versus the need to

(1) support an ever-increasing number of architectures,
(2) deploy over increasingly large-scale systems consisting of multiple such architectures, and
(3) deal with a mixture of shared- and distributed-memory connectivity across compute units.

While 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 humble code, once compiled, should perform on par with expert code, thus turning humble programmers into heroes.

A recent study shows the humble ALP/GraphBLAS outperforms the state of the art for three graph and sparse matrix algorithms [MAY22a]. ALP/Pregel exposes a vertex-centric programming model and compile-time translates vertex-centric programs into ALP/GraphBLAS, while benefiting from its shared- and distributed-memory auto-parallelisation and other optimisations. This work demonstrates that a vertex-centric strongly connected components algorithm implemented using ALP/Pregel obtains up to 17.5× speedup on a dual-socket Intel Xeon machine. Similarly, an auto-parallelised vertex-centric PageRank-like algorithm obtains speedups up to 17.8×, which confirms that ALP/Pregel programs scale. Furthermore, comparing the vertex-centric program against a highly-optimised canonical PageRank algorithm results in faster sequential execution in twelve out of thirteen cases, and speedups of up to 8.99×. In shared-memory parallel execution, the ALP/Pregel algorithm is fastest in five out of thirteen cases, with speedups up to 10×. 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 optimised canonical solutions.

7.1. **Outlook.**

Different groups of humble programmers may prefer different humble programming interfaces, and 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. Also ideally, only a few humble programming interfaces with corresponding optimised software stacks are required. This paradox can be resolved by translating many useful humble programming interfaces into a just a few optimised 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 validates the notion of supporting multiple humble programming models on top of a single software stack, demonstrates this is possible without significant performance overheads, and solves the software bottleneck as it frees hero programmers to address optimisation and portability concerns for the few fundamental software stacks only– in this case, the algebraic programming stack.

Even so, this paper does not argue that ALP should be the sole such fundamental programming model: it remains an open question how much of the general-purpose programming demands may be covered by ALP, as well as how many popular humble programming interfaces may be implemented on top of ALP without incurring significant performance overheads. Instead, we may more humbly argue that the future of software would be vastly improved if the industry and research communities explore more novel 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 generally are naturally loosely-coupled. At any point in time during execution, not all vertices are required to be in the same round– vertices whose round-i messages have been received can immediately continue with round $i+1$, regardless of whether some vertices are still pending to execute the ith computation phase or whether they are waiting on incoming round-i messages. This insight can be applied recursively to realise pipelines of arbitrary length, bounded only by the underlying graph structure and the available memory for caching in-flight messages.

A Pregel program running on the source vertex of a line graph can progress arbitrary round numbers ahead of the other vertices, for example, thus leading 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 in 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. [MAY22a, MAY22b] realises overlap between communication and computation phases of a single round. By introducing buffers for incoming messages that same approach may also enable overlapping rounds in computation discussed here, and will be the next step to take in enhancing 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. Instead, the aim is to provide such comparison in future work, comparing also the effects of loosy-coupled execution.

Before this, a first aim brings 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 to avoid having to cache incoming messages, thus both saving memory and increasing data reuse, seems a most promising direction.

7.3. **Beyond ALP/Pregel.** While this paper shows that realising the Pregel humble programming model on top of ALP is possible and thus may benefit of 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 automatic translate into ALP. For MapReduce, for example, the following insights may automatically translate MapReduce programs into ALP:

- a one-to-one map phase again corresponds to the element-wise lambda function;
- one-to-none or one-to-many maps may be realised by the parallel I/O mode introduced by ALP/GraphBLAS [YDNNS20]; and
- a reduce phase may be realised 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 again an SpMV multiplication, and
  (4) transforming the resulting vector of values back into a vector of key-value pairs.

This approach, ALP/MapReduce, 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 31st August 2022.

[Alg21b] Algebraic Programming. ALP/GraphBLAS release v0.6. `https://gitee.com/CSL-ALP/graphblas`, 2021. Last retrieved 31st August 2022.

[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, Aydın 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, page 282–297, New York, NY, USA, 2021. Association for Computing Machinery.

[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)*, pages 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, July 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. In *Generic programming*, pages 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)*, pages 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*, pages 211–222, 2007.

[LM11] Amy N Langville and Carl D Meyer. Google's PageRank and beyond. In *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*, pages 135–146, 2010.

[MAY22a] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman. Design and implementation for non-blocking execution in GraphBLAS: tradeoffs and performance. Accepted under minor revision, 2022.

[MAY22b] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman. Nonblocking execution in GraphBLAS. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 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*, pages 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)*, pages 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 $O(\log n)$ 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. Preprint, 2020.

[ZAB20] Yongzhe Zhang, Ariful Azad, and Aydın 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.