Efficient handling of sparse vectors for parallel nonblocking execution in GraphBLAS

Aristeidis Mastoras[0000−0002−5235−8499] and Albert-Jan N. Yzelman[0000−0001−8842−3689]

Computing Systems Laboratory, Huawei Zurich Research Center, Switzerland

Abstract. GraphBLAS allows the expression of algorithms in the language of linear algebra, and it aims at performing automatic optimization and parallelization. Recent work shows that parallel nonblocking execution outperforms the corresponding blocking execution up to 4.11×, by reusing data in cache. However, the presented design, implementation, and evaluation focus on dense vectors. In this work, we present design and implementation extensions for efficient handling of sparse vectors, and to reduce the overhead, we propose compile-time and run-time optimizations. The evaluation shows mixed results and promising speedups up to 1.72× for the performance of nonblocking over blocking execution.

Keywords: GraphBLAS· nonblocking execution · lazy evaluation · sparse vectors · dense vectors · loop fusion · loop tiling

1 Introduction

GraphBLAS [5] is a recent standard for the expression of algorithms in the language of linear algebra. The C API specification defines two execution modes: blocking execution and nonblocking execution [5]. The *blocking* mode guarantees that the computation is completed and the result is written to memory when the function of an operation returns. The *nonblocking* mode allows an operation to return although the result has not been computed yet. Thus, the nonblocking mode may delay the execution and perform optimizations.

Lazy evaluation is the key idea in nonblocking execution, and computations are performed only when they are required for the sound execution of a program. Mastoras et al. [19, 20] present the design and implementation for nonblocking execution, and the evaluation shows speedups up to 4.11× over the corresponding blocking execution. In particular, GraphBLAS programs are memory-bound and the obtained speedups are the result of better cache utilization. However, the presented design and implementation covers only dense vectors, and none of the considered algorithms used in the evaluation results in sparse vectors [19, 20].

In this paper, we extend the work of Mastoras et al. [19] by presenting a preliminary design and implementation for efficient handling of sparse vectors. An evaluation of Pregel Pagerank, by using a set of various matrices selected from the SuiteSparse Matrix Collection [7], shows mixed results for the performance of nonblocking compared to blocking execution. In particular, the speedups range from 0.18× to 1.72×, confirming the importance of the nonblocking execution.

2 Background

Mastoras et al. [19] present the design and implementation for fully automatic nonblocking execution of level-1 and level-2 operations. The design may be integrated into any GraphBLAS implementation and was implemented in the open-source [1] C++ ALP/GraphBLAS [26] v0.1. The presented extensions for efficient handling of sparse vectors refer to v0.7 of the same implementation.

2.1 Nonblocking execution in ALP/GraphBLAS

Lazy evaluation enables loop fusion and loop tiling at run-time, and dynamic data dependence analysis identifies operations that share data and adds them as stages of the same pipeline. Operations grouped together may be executed in parallel and reuse data in cache. The nonblocking execution is fully dynamic, since the optimizations are performed at run-time and the operations may have arbitrary control flow. The nonblocking execution is fully automatic, since all performance parameters, e.g., tile size, are selected based on an analytic model [19].

```
void operation(Vec x, ...) {
  auto func = [x, ...](size_t low, size_t up) {
    for (size_t i = low; i < up; ++i)
      x[i] = ...;
  }
  storeFunc(func);
}

...

for (i = 0; i < N; i += T)
  for (auto func = begins(); func != ends(); ++func)
    (*func)(i, min(i + T, N));
```

Fig. 1: Pseudo-code for lazy evaluation employed by the nonblocking execution.

Figure 1 shows a typical GraphBLAS operation with a main loop that iterates over the elements of the containers, i.e., vectors and matrices. For the nonblocking execution, the loop is not executed when the function is invoked. Instead, the loop is added into a lambda function that corresponds to a stage of a pipeline, and it is stored and executed later. The execution of a pipeline is performed by two nested loops. The outer loop has the same bounds with the operations added into the pipeline, and the iteration space is partitioned into tiles of size `T` to perform the loop tiling optimization. The inner loop iterates over all the stages of the pipeline to perform the loop fusion optimization. A lambda function is invoked for each tile of a stage and executes a subset of iterations for the initial loops determined by the bounds of the tile. In the case of dense vectors, the internal loop of the lambda functions iterates over all the elements of the vectors. However, iterating over only the nonzeroes of a sparse vector requires additional mechanisms.

2.2 Vector representation in ALP/GraphBLAS

Vectors may be either sparse or dense. In the case of dense vectors, each operation accesses all the elements. However, to efficiently handle sparsity, it is necessary to maintain the coordinates of the nonzeroes, such that operations access only the nonzeroes. Hence, each vector is represented with the following data:

- an unsigned integer `n` that stores the size of the vector;
- an unsigned integer `nnz` that stores the number of nonzeroes in the vector;
- a boolean array, `assigned`, of size `n` that indicates if the element of a coordinate is a nonzero; and
- an unsigned integer array, `stack`, that represents a stack and stores the coordinates of the assigned elements.

A vector is dense when the number of nonzeroes is equal to the size of the vector, i.e., `nnz = n`. The stack and the `assigned` array are used only when accessing a sparse vector. For an empty vector, `nnz = 0`, all the elements of `assigned` are initialized to `false`, and the stack is empty. The assignment of the i-th element of a vector implies that: `stack[nnz] = i`; `assigned[i] = true`; and `nnz` is incremented by 1. Therefore, the coordinates of the nonzeroes are not sorted; they are pushed to the stack in an arbitrary order. Iterating over the nonzeroes of a sparse vector is performed via the stack. This data structure is also known as a sparse accumulator (SPA) [10].

2.3 Problem statement

The internal representation of a vector is sufficient to correctly and efficiently handle sparse vectors for serial execution. However, this is not the case for multi-threaded execution, since simultaneous assignments of vector elements may cause data races. Protecting the stack and the counter of nonzeroes with a global lock is a trivial solution that leads to significant performance degradation. Therefore, it is necessary to design a different mechanism that is tailored to the needs of the nonblocking execution and exploits any information about accesses of elements by different threads.

3 Efficient handling of sparse vectors

We present the design for efficient handling of sparse vectors in parallel nonblocking execution provided under the ALP/GraphBLAS v0.7 implementation. The design introduces the local coordinates mechanism, i.e., a set of local views for the coordinates stored in the global stack. Each local view includes the coordinates of the nonzeroes for a tile of iterations. To avoid the high overhead of locking every time a new element is assigned to a vector, each thread accesses its own local coordinates and any update to the sparsity pattern of a vector is performed in the local view. The local coordinates mechanism requires initialization of the local views before the execution of the pipeline and update of the global stack with the new nonzeroes after the execution of the pipeline.

3.1 Local coordinates mechanism

The local coordinates mechanism requires some additional data for each of the `num_tiles` tiles of a vector:

- an unsigned integer array `local_nnz` that stores the number of nonzeroes for each local view, which are read from the global stack during initialization;
- an unsigned integer array `local_new_nnz` that stores the number of nonzeroes assigned to each local view during the execution of a pipeline; and
- a set of unsigned integer arrays, `local_stack`, that represent local stacks and store the local coordinates, i.e., each array corresponds to a local view.

The local coordinates mechanism relies on the functions shown in Figure 2. The local views are initialized via `asyncSubsetInit`, while `asyncSubset` and `asyncJoinSubset` read and update the state of the local view, respectively. Pushing the local coordinates to the global stack is performed via `joinSubset`. None of these functions uses locks, and to avoid data races, `joinSubset` updates the global stack based on the prefix-sum computation for the number of new nonzeroes performed by `prefixSumComputation`.

Figure 3 shows an example of how the lambda functions use the local coordinates mechanism for a simple program that invokes `set` and `foldl`. The state of the local view is read for each vector accessed in an operation by invoking `asyncSubset`. The sparsity pattern may be updated only for the output vector, and thus `asyncJoinSubset` is invoked only for the output vector of `foldl`. Operations consider the dense and the sparse case, and the executed path is determined at run-time based on the sparsity pattern of the local coordinates.

Figure 4 shows the four steps required for the execution of a pipeline. In particular, the local view is initialized for each tile of each vector accessed in the pipeline, and then the pipeline is executed. Once the execution is completed, the local views may contain a number of new nonzeroes that are pushed to the global stack by `joinSubset`. Before this step, it is necessary to perform the prefix-sum computation for the number of new nonzeroes of each local view. All these steps may be executed in parallel for different tiles of the vectors as shown with the OpenMP [22] directives. The prefix-sum computation is parallelized internally.

3.2 Performance tradeoffs

The design of the local coordinates mechanism avoids the overhead of locks but requires initialization of the local views and the update of the global stack. Hence, the local coordinates mechanism should be used only when necessary. The sparsity pattern of input vectors cannot change during the execution of an operation, and thus it is safe to use the global coordinates. However, the input vector of an operation may be the output vector of another operation in the same pipeline. To correctly handle this situation, there exist two design options. First, each operation uses the local coordinates mechanism for all output and input vectors accessed in a pipeline as shown in Figure 2. Second, each operation uses the local coordinates mechanism for the output vector

```
void asyncSubsetInit(size_t low, size_t up) {
  ...
  c = 0;
  if (up - low < nnz)
    // Iterates over the elements of the tile and checks if they are assigned
    for (i = low; i < up; ++i)
      if (assigned[ i ])
        local_stack[ c++ ] = i - low;
  else
    // Iterates over the nonzeroes and checks if they are part of the tile
    for (i = 0; i < nnz; ++i)
      if (stack[ i ] >= low && stack[ i ] < up)
        local_stack[ c++ ] = stack[ i ] - low;

  local_nnz[tile_id] = c; // Initializes the local counter of nonzeroes
  local_new_nnz[tile_id] = 0; // and the local counter of new nonzeroes
}

Coordinates asyncSubset(size_t low, size_t up) {
  Coordinates ret;
  ...
  // Initializes the total number of nonzeroes when asyncSubset is invoked
       for each operation; the new nonzeroes are 0 when asyncSubset is invoked
        for the first operation of the pipeline
  ret.nnz = local_nnz[tile_id] + local_new_nnz[tile_id];
  return ret;
}

void asyncJoinSubset(Coordinates &subset) {
  ...
  // Updates the number of new nonzeroes for this tile such that the counter
       is up to date after the execution of each operation in a pipeline
  local_new_nnz[tile_id] = subset.nnz - local_nnz[tile_id];
}

void prefixSumComputation() {
  ...
  // In practice, the prefix sum is performed in parallel
  pref_sum[0] = nnz + local_new_nnz[0];
  for (i = 1; i < num_tiles; ++i)
    pref_sum[i] = pref_sum[i - 1] + local_new_nnz[i];

  // The prefix sum of each tile includes the initial number of nonzeroes due
        to the initialization above; the new total number of nonzeroes is the
       prefix sum of the last tile and is updated by a single thread
  nnz = pref_sum[num_tiles - 1];
}

void joinSubset(size_t low, size_t up) {
  ...
  // The new nonzeroes of each tile are written to a position of the stack
       determined by the prefix sum, which includes the number of new
       nonzeroes which is deducted to compute the starting position
  size_t pos = pref_sum[tile_id] - local_new_nnz[tile_id];
  for (k = 0; k < local_new_nnz[tile_id]; ++k)
    stack[pos++] = local_stack[local_nnz[tile_id] + k] + low;
}
```

Fig. 2: Pseudo-code for the local coordinates mechanism.

and the global coordinates for the input vectors. In this case, the invocation of `asyncSubsetInit` and `joinSubset` in the lambda functions must essentially replace that of `asyncSubset` and `asyncJoinSubset`, respectively. Since most

```
int main() {
  ...
  set(x, c);      // x[i] = c for all i; x is a vector and c is a scalar
  foldl(y, x, f); // y = f(y, x), x and y are vectors
  ...
}

void set(Vec x, Scal c) {
  auto func1 = [x, c](size_t low, size_t up) {
    Coor local_cr_x = x.getCoor().asyncSubset(low, up);
    ...
    if (dense)
      for (i = low; i < up; ++i)
        x[i] = c;
    else
      for (i = 0; i < local_cr_x.nnz(); ++i) {
        x[i + low] = c;
      }
    x.getCoor().asyncJoinSubset(local_cr_x);
  }
  storeFunc(func1);
}

void foldl(Vec y, Vec x, Op f) {
  auto func2 = [y, x, f](size_t low, size_t up) {
    Coor local_cr_y = y.getCoor().asyncSubset(low, up);
    Coor local_cr_x = x.getCoor().asyncSubset(low, up);
    ...
    if (dense)
      for (i = low; i < up; ++i)
        y[i] = f(y[i], x[i]);
    else
      for (i = 0; i < local_cr_y.nnz(); ++i) {
        ...
        y[i + low] = f(y[i + low], x[i + low]);
      }
    y.getCoor().asyncJoinSubset(local_cr_y);
  }
  storeFunc(func2);
}
```

Fig. 3: Pseudo-code for lazy evaluation that considers sparse vectors.

GraphBLAS operations have a single output, which is either a vector or a scalar, this design option results in using the local views for at most as many vectors as the number of stages in a pipeline. Therefore, the design option that leads to the best performance depends on the pipelines created for each algorithm.

4 Optimizations for dense vectors

To improve the performance of nonblocking execution, it is crucial to avoid the usage of the local views when the vectors are dense. It is possible to determine whether a vector is dense based on compile-time information from descriptors and run-time analysis. The first one implies zero run-time overhead, but the descriptors must be provided by the user. There exist two main differences between the compile-time information from descriptors and the run-time analysis. First, descriptors may apply to all vectors of an operation, whereas the run-time

```
#pragma omp parallel for
for (i = 0; i < N; i += T)
  for( auto coor = beginc(); coor != endc(); ++coor )
    (**coor).asyncSubsetInit(i, min(i + T, N));

#pragma omp parallel for
for (i = 0; i < N; i += T)
  for (auto func = begins(); func != ends(); ++func)
    (*func)(i, min(i + T, N));

for( auto coor = beginc(); coor != endc(); ++coor )
    (**coor).prefixSumComputation();

#pragma omp parallel for
for (i = 0; i < N; i += T)
  for( auto coor = beginc(); coor != endc(); ++coor )
    (**coor).joinSubset(i, min(i + T, N));
```

Fig. 4: Execution of the lazily evaluated operations for sparse vectors, including the initialization of the local coordinates.

analysis applies to each individual vector of an operation. Second, descriptors refer to the vectors of a specific operation, whereas the run-time analysis refers to the state of a vector before the execution of a pipeline.

4.1 Compile-time descriptors

ALP/GraphBLAS provides a set of descriptors that may be combined using bitwise operators. A descriptor is passed to an operation and indicates information about some or all of the inputs. Three of these descriptors are the following:

- `dense` to indicate that all input and output vectors are structurally dense before the invocation;
- `structural` that ignores the values of the mask and uses only its structure, i.e., the i-th element evaluates to true if any value is assigned to it; and
- `invert_mask` that inverts the mask.

The `dense` descriptor may affect both correctness and performance, and `structural` and `invert_mask` affect only the correctness of an operation. These three descriptors may be used to perform optimizations for the local coordinates mechanism. In particular, if the dense descriptor is provided, it implies that all the vectors accessed in an operation are dense before the invocation. Therefore, an operation can safely iterate over all the elements of the vectors without using either the global or the local coordinates.

One exception is an out-of-place operation that receives a mask, since the dense descriptor itself does not guarantee that all the elements of a dense mask evaluate to true. Therefore, a dense output vector may become sparse once the computation is completed. That is, the output vector becomes empty in the beginning of the operation, and then each of its coordinates may be assigned depending on whether the corresponding element of the mask evaluates to true

or not. Reading the elements of a mask does not require usage of the local coordinates when the dense descriptor is given.

However, to avoid the usage of the local coordinates for the output vector of an out-of-place operation that receives a mask, both the `structural` and the `invert_mask` descriptors should be provided in addition to the `dense` descriptor.

4.2 Run-time analysis

The run-time analysis for dense vectors relies on a property of ALP/GraphBLAS. A vector that is already dense before the execution of a pipeline may become sparse during the execution of the pipeline only when the pipeline contains an out-of-place operation, for example, `grb::set` or the `grb::clear` operation that makes the vector empty. The current design for nonblocking execution in ALP/GraphBLAS allows pipelines that include an out-of-place operation but does not allow pipelines that include the `grb::clear` operation.

The nonblocking execution relies on run-time analysis to determine whether a vector is already dense, only when the `dense` descriptor is not given by the user. For each already dense vector of a pipeline, neither the global nor the local coordinates are used unless the vector is the output of an out-of-place operation. Thus, the overhead of the local coordinates mechanism is completely avoided.

5 Evaluation

We conducted a preliminary evaluation for nonblocking execution on Pregel Pagerank that uses vertex inactivation [25] and results in sparse vectors. We used various matrices selected from the SuiteSparse Matrix Collection [7]. The evaluation includes experimental results for the serial blocking, parallel blocking and parallel nonblocking execution of ALP/GraphBLAS [26]. The source code of ALP/GraphBLAS is publicly available [1] together with Pregel Pagerank.

We used a dual-socket system with two processors Intel Xeon Gold 6238T and a total of 44 cores at a clock frequency of 1.90 GHz, and we disabled both hyper-threading and turbo boost. The main memory is 192 GB, L1 cache is 32 KB, L2 cache is 1 MB, and L3 cache is 30.25 MB. The source code was compiled with g++ 13.3.0 and using performance flags on Ubuntu 24.04.2 LTS.

The serial execution uses one core, and we set 44 threads for both parallel blocking and nonblocking execution. For the nonblocking execution, the number of threads and the tile size are automatically selected by an analytic model [19]. We set thread-to-core affinities for all implementations. The presented speedups are based on an average of 30 measurements for each experiment, and the standard deviations are less than 0.1% of the average execution times.

Table 1 presents the speedups for the parallel blocking execution and the parallel nonblocking execution over the serial execution. We notice that none of the two execution modes outperforms the other one for all matrices. The blocking execution achieves the highest speedup over nonblocking execution for the `parabolic_fem` matrix, and the nonblocking execution achieves the highest speedup over the blocking execution for the `G2_circuit` matrix.

Matrix	Size	Nonzeroes	Serial	Blocking	Nonblocking
gyro_m	17'361	340'431	0.63 ms	**1.06**×	0.60×
vanbody	47'072	2'329'056	2.87 ms	3.74×	**5.82**×
Dubcova3	146'689	3'636'643	2.13 ms	2.78×	**3.81**×
G2_circuit	150'102	726'674	2.18 ms	2.13×	**3.67**×
hood	220'542	9'895'422	9.20 ms	6.58×	**10.46**×
offshore	259'789	4'242'673	7.95 ms	5.00×	**8.07**×
inline_1	503'712	36'816'170	44.41 ms	10.54×	**16.94**×
bundle_adj	513'351	20'207'907	40.60 ms	**8.45**×	5.76×
parabolic_fem	525'825	3'674'625	1.07 ms	**1.28**×	0.24×
apache2	715'176	4'817'870	13.94 ms	**3.58**×	2.47×
Emilia_923	923'136	40'373'538	32.30 ms	**6.86**×	4.75×
ecology2	999'999	4'995'991	2.08 ms	**1.76**×	0.38×
Serena	1'391'349	64'131'971	75.52 ms	8.24×	**8.46**×
G3_circuit	1'585'478	7'660'826	26.83 ms	**3.84**×	3.53×
Bump_2911	2'911'419	127'729'899	72.86 ms	**6.94**×	6.37×
wikipedia-20070206	3'566'907	45'030'389	115.30 ms	**12.14**×	6.95×
Queen_4147	4'147'110	316'548'962	45.19 ms	**6.86**×	3.52×
adaptive	6'815'744	27'248'640	15.90 ms	**3.15**×	1.02×
uk-2002	18'520'486	298'113'762	432.78 ms	8.48×	**8.74**×
road_usa	23'947'347	57'708'624	786 ms	6.21×	**10.41**×

Table 1: Speedups over serial blocking execution for the parallel blocking and parallel nonblocking execution of Pregel Pagerank by using various matrices. Serial corresponds to the average execution time for one iteration of the algorithm.

6 Related Work

The nonblocking mode, defined by the GraphBLAS standard [5], is implemented to various extents by SuiteSparse::GraphBLAS [6] and ALP/GraphBLAS [26]. Nonblocking mode in SuiteSparse:GraphBLAS is primarily exploited by postponing operations that should update sorted data structures for (sparse) vectors and/or sparse matrices. One straightforward example is adding single elements to some sparse container x. Instead of inserting new elements one-by-one into sparse data structures, a small stack maintains the elements that need to be inserted. If the operations that are executed on x do not require its elements to be sorted, then the operations employ the incomplete sorted storage and the unsorted new elements in the small stack. This works for operations such as sparse matrix–vector multiplication by exploiting the associativity of the addition operation. On the other hand, if a requested operation demands the container to be sorted, then any pending elements are first inserted in bulk. A call to `grb::wait` may force any pending insertions to complete in bulk as well.

Similar frameworks to GraphBLAS, that are tailored to numerical linear algebra applications, include Eigen [11] and the Sparse BLAS [2,9,23]. Eigen per-

forms fusion based on expression templates and thus achieves an effect similar to nonblocking execution in ALP/GraphBLAS but does not optimize for sparsity in vectors. The current Sparse BLAS standard [9] as well as its predecessor [23] define primitives for sparse vector operations but do not allow nonblocking execution. A currently ongoing effort for an updated Sparse BLAS aims at defining an opaque container interface that allows for nonblocking execution. However, the proposed API focuses on transparent C++ views over external CRS or CCS matrices, and at present, does not define primitives for sparse vectors [2].

Taking a graph-centric view, nonblocking execution using sparse vectors amounts to tiling through a number of successive graph operations that each only touches a subset of vertices. Doekemeijer and Varbanescu introduce the concept of incremental scheduling for graph processing frameworks that operate on a succession of increasingly sparse vertex subsets [8]. However, frameworks that perform incremental scheduling may not necessarily tile through successive operations. For example, Ligra [24] optimizes incremental scheduling via direction optimization [4], which is also supported by ALP/GraphBLAS [26]. This optimization is orthogonal to tiling and, in linear algebraic terms, affects only the sparse matrix–vector operation. Pregel [18] is the most prominent graph processing framework that automatically exploits tiling through vertex subsets, and ALP/GraphBLAS enables this optimization through a lens of generalized linear algebra. In this context, Yzelman [25] notes that graph-centric and matrix-centric views coincide and allow for equivalent optimizations.

There exist vertex-centric graph processing systems inspired by Pregel that do not perform tiling. For example, Giraph [12] is based completely on Hadoop's MapReduce implementation [3] and executes Pregel programs round-by-round. Heidari et al. [13] introduce a taxonomy of graph processing frameworks that includes asynchronous coordination between graph partitions. A total of 27 out of 64 frameworks included in the survey are classified as asynchronous, 25 of which perform asynchronous coordination employing a shared-memory mode and thus may include optimizations similar to those of ALP/GraphBLAS. Among those, the more prominent ones are GraphChi [16], GraphLab [17], Ligra [24], and Galois [14, 15, 21]. GraphChi represents partitions by files, which are updated and read asynchronously. Similar to GraphBLAS [5], GraphChi includes an optimization for iterative sparse matrix–vector applications that allows user-defined plus and times operations, but it does not allow incremental scheduling. GraphLab and Ligra allow incremental scheduling, but neither framework tiles through successive rounds of vertex updates. Galois allows the expression of iterative schedules while potentially performing dynamic tiling through successive operations, making it similar to our work. However, Galois uses speculative parallelism with roll-backs supported through negation of too-optimistically applied operations, and it allows tiling based on commutativity properties. These are examples of exploiting algebraic information on the program specification for optimization, which is common with our ALP-based [25] approach.

The execution on distributed systems is another important aspect to consider when comparing established graph frameworks such as Pregel [18], with

ALP/GraphBLAS [26]. In particular, ALP/GraphBLAS provides execution on both shared- and distributed-memory systems, but the nonblocking execution is currently available only on shared-memory systems. Moreover, a fault tolerance mechanism has yet to be realized with ALP/GraphBLAS.

7 Conclusion

Recent work for nonblocking execution in GraphBLAS focuses on algorithms that result in dense vectors and shows performance improvement as a result of data locality optimizations. In this work, we present the necessary extensions for efficient handling of sparse vectors for nonblocking execution. The main challenge of handling sparse vectors for multi-threaded execution is the data races caused by concurrent updates on the sparsity pattern. The presented design avoids the high overhead of trivial solutions that rely on locks, and it exploits compile-time and run-time information to perform optimizations that improve the overall performance. A preliminary evaluation shows promising speedups for nonblocking over blocking execution by using various matrices for Pregel Pagerank.

References

1. ALP/GraphBLAS (2021), https://github.com/Algebraic-Programming/ALP, retrieved on June 6, 2025
2. Abdelfattah, A., Ahrens, W., Anzt, H., Armstrong, C., Brock, B., Buluc, A., Busato, F., Cojean, T., Davis, T., Demmel, J., Dinh, G., Gardener, D., Fiala, J., Gates, M., Haider, A., Imamura, T., Lara, P.V., Moreira, J., Li, S., Luszczek, P., Melichenko, M., Moeira, J., Mokwinski, Y., Murray, R., Patty, S., Peles, S., Ribizel, T., Riedy, J., Rajamanickam, S., Sao, P., Shantharam, M., Teranishi, K., Tomov, S., Tsai, Y.H., Weichelt, H.: Interface for Sparse Linear Algebra Operations (2024), https://arxiv.org/abs/2411.13259
3. Apache Software Foundation: Apache hadoop. https://hadoop.apache.org/, retrieved on June 6, 2025
4. Beamer, S., Asanović, K., Patterson, D.: Direction-Optimizing Breadth-First Search. Scientific Programming **21**(3-4), 702694 (2013)
5. Brock, B., Buluç, A., Kimmerer, R., Kitchen, J., Kumar, M., Mattson, T., McMillan, S., Moreira, J., Pelletier, M., Welch, E.: The GraphBLAS C API Specification, version 2.1 (2023), https://graphblas.org/docs/GraphBLAS_API_C_v2.1.0.pdf, retrieved on June 6, 2025
6. Davis, T.A.: Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. ACM Transactions on Mathematical Software **45**(4), 44:1–44:25 (Dec 2019)
7. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software **38**(1), 1:1–1:25 (Nov 2011)
8. Doekemeijer, N., Varbanescu, A.L.: A Survey of Parallel Graph processing Frameworks. Parallel and Distributed Systems Report Series PDS-2014-003, Delft University of Technology, Delft, Netherlands (2014)
9. Duff, I.S., Heroux, M.A., Pozo, R.: An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. ACM Transactions on Mathematical Software **28**(2), 239–267 (Jun 2002)

10. Gilbert, J.R., Moler, C., Schreiber, R.: Sparse Matrices in MATLAB: Design and Implementation. SIAM Journal on Matrix Analysis and Applications **13**(1), 333–356 (1992)
11. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010), http://eigen.tuxfamily.org, retrieved on June 6, 2025
12. Han, M., Daudjee, K.: Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. Proceedings of the VLDB Endowment **8**(9), 950–961 (May 2015)
13. Heidari, S., Simmhan, Y., Calheiros, R.N., Buyya, R.: Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. ACM Computing Surveys **51**(3) (Jun 2018)
14. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 211–222. PLDI '07 (2007)
15. Kulkarni, M.V.: The Galois System: Optimistic Parallelization of Irregular Programs. Ph.d. dissertation, Cornell University (2008)
16. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. p. 31–46. OSDI'12 (2012)
17. Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E., Hellerstein, J.: GraphLab: A New Framework For Parallel Machine Learning (2014), https://arxiv.org/abs/1408.2041
18. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. p. 135–146. SIGMOD '10 (2010)
19. Mastoras, A., Anagnostidis, S., Yzelman, A.N.: Design and Implementation for Nonblocking Execution in GraphBLAS: Tradeoffs and Performance. ACM Transactions on Architecture and Code Optimization **20**(1), 6:1–6:23 (Nov 2022)
20. Mastoras, A., Anagnostidis, S., Yzelman, A.N.: Nonblocking execution in GraphBLAS. In: 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 230–233 (2022)
21. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. p. 456–471. SOSP '13 (2013)
22. OpenMP Architecture Review Board, Version 6.0: OpenMP Application Program Interface (2024), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf, retrieved on June 6, 2025
23. Remington, K., Pozo, R.: The NIST Sparse BLAS (v0.9), National Institute of Standards and Technology, https://math.nist.gov/spblas.original.html, retrieved on June 6, 2025
24. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 135–146. PPoPP '13 (2013)
25. Yzelman, A.N.: Humble heroes. Communications of Huawei Research **6**, 146–170 (June 2024)
26. Yzelman, A.N., Di Nardo, D., Nash, J.M., Suijlen, W.J.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation (2020), http://albert-jan.yzelman.net/PDFs/yzelman20.pdf, retrieved on June 6, 2025