Considerations for a Distributed GraphBLAS API

Benjamin Brock*, Aydın Buluç†*, Timothy G. Mattson‡, Scott McMillan§,
José E. Moreira¶, Roger Pearce‖, Oguz Selvitopi†, Trevor Steil**

* EECS Department, University of California, Berkeley, CA
† Computational Research Department, Lawrence Berkeley National Laboratory, Berkeley, CA
‡ Parallel Computing Labs, Intel, Hillsboro, OR
§ Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA
¶ IBM Thomas J. Watson Research Center, Yorktown Heights, NY
‖ Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA
** School of Mathematics, University of Minnesota, Minneapolis, MN

***Abstract*—The GraphBLAS emerged from an international effort to standardize linear-algebraic building blocks for computing on graphs and graph-structured data. The GraphBLAS is expressed as a C API and has paved the way for multiple implementations. The GraphBLAS C API, however, does not define how distributed-memory parallelism should be handled. This paper reviews various approaches for a GraphBLAS API for distributed computing. This work is guided by our experience with existing distributed memory libraries. Our goal for this paper is to highlight the pros and cons of different approaches rather than to advocate for one particular choice.**

I. Introduction

There are multiple ways to represent graphs. One approach exploits the connection between graphs and sparse matrices; in which graph algorithms are expressed in the language of linear algebra [10]. The GraphBLAS forum was formed to standardize an API for this approach to Graph algorithms. The resulting specification is called the *GraphBLAS* which is defined by a pair of documents: a mathematical specification [9] and a C API [7].

The GraphBLAS C API is mature and has been used to support multiple implementations of the GraphBLAS. The current API (version 1.3.1) encourages parallel algorithms embedded "inside" the GraphBLAS functions, but it says nothing about explicit management of parallelism. That API is adequate for multithreaded libraries [2], but it does not address issues raised by more general parallelism.

In this position paper, we start with a discussion of the changes needed in the GraphBLAS C API to support general parallelism. We then discuss the complex case of distributed memory parallelism, suggesting a few specific options. We do not, however, choose any of these options. Our goal at this point is to define our options and foster a discussion about distributed memory parallelism with the GraphBLAS community; a critical discussion as we anticipate creating a GraphBLAS specification that supports distributed memory parallelism over the next year.

II. Parallelism in GraphBLAS

We begin by establishing the terminology we will use in this paper. A program defines operations which constitute the *work* associated with the program. The program is compiled and linked to create an *executable*. An instance of a running executable and the resources needed to support it are a *job*. The work of a program can be decomposed into one or more sets of *tasks* that can execute concurrently. Given parallel hardware, these concurrent tasks can execute in parallel thereby letting the work defined by a job complete in less time.

A parallel API defines mechanisms to exploit the concurrency in a program to support execution on parallel hardware. Fundamental issues addressed by any parallel API include:

- Management of a collection of execution agents that make forward progress in parallel.
- Mapping the work of a program (the tasks) onto the execution agents.
- Mapping of data onto the memories associated with the execution agents.
- Coordination (i.e., communication and synchronization) between execution agents.

A program that supports parallel execution of a job is called a *parallel program*. For a scalable parallel program, the job should execute *efficiently* meaning they should effectively utilize the processing elements of the parallel system. Furthermore, the programs must execute correctly meaning they avoid numerical instabilities and concurrency errors (data races and deadlocks). We will consider two broad classes of parallel systems: shared memory multiprocessor systems and distributed memory multicomputer systems.

For a shared memory multiprocessor system, the execution agents are threads so the associated parallel program is called a *multithreaded* program. The set of threads execute in a single address space. In most modern shared memory multiprocessor systems, sharing of data between threads is managed by a cache coherency system. This greatly simplifies the effort a programmer must expend to map data between execution agents (though for best performance, the nonuniform memory features of the system must be taken into account which can be quite complicated). Coordination in these systems involves ordering potentially conflicting loads and stores to shared addresses (*synchronization*).

On a distributed memory system, the execution agent is typically a *process*. Processes are gathered into groups called *process groups*. Each process has its own memory, which works well with systems which lack physical shared memory (such as "shared-nothing" clusters). Lacking a shared address space, the challenge with these systems is how to decompose the data associated with a job into partitions that can be mapped onto a group of processes. The processes communicate by passing messages between each other, either explicitly for message passing APIs such as MPI (Message Passing Interface), or implicitly behind a more abstract put/get API in a partitioned global address space (PGAS) model. Messaging occupies a namespace of sorts called a communication context. This lets programmers organize messages into distinct sets that do not conflict. The most commonly used message passing library is MPI. In MPI, the process group and context for messages are combined into a single object called the *communicator*. Any library built on top of MPI must decide how to manage its communicators. This is a key aspect of our design of a GraphBLAS API for distributed memory systems.

The current GraphBLAS specification is suitable for multithreaded programs where the threads are hidden from the programmer inside the individual GraphBLAS methods. In essence, the multithreaded and sequential APIs are nearly the same. Due to the complexities of partitioning the data in a job and the need to manage communicators, however, the API for a distributed memory system cannot make parallelism completely abstract. By necessity, we need to define an explicitly parallel API for distributed memory systems.

III. Distributed GraphBLAS: Data Distribution

The GraphBLAS API presents objects as opaque data structures. This lets implementors of GraphBLAS libraries hide the complexity of data structures and how they map onto different systems. It is important to continue with this approach as we move into a distributed GraphBLAS API. This implies that we will need to adopt a Partitioned Global Address Space (PGAS) approach which views GraphBLAS objects as distributed data types. An important design decision with PGAS systems is how indices of array objects handled.

In libraries where users manage the explicit details of their data structures, the programmer must work in terms of two index spaces: local indices and global indices. The local indices correspond to elements of an object that physically reside in the memory of a particular process. The global indices are the "mathematically defined" indices for the full array object. When working with such libraries the user must manage the mapping between local and global indicies.

Given our goal of presenting users with opaque data structures, we believe it is important to avoid local vs. global index mappings. Users should only interact with global indices with any mapping onto local indices occurring inside GraphBLAS functions.

IV. Distributed GraphBLAS: Communicators

A user writes a parallel program for execution on a distributed system. This program combined with needed resources (such as the data set) constitutes a job. Inside the program, groups of processes interact by exchanging messages. A communicator is used to prevent conflicts between different process groups or communication contexts.

When a distributed data structure is created, it is associated with a specific communicator. Only the processes of a particular group that share a communication context can access that data structure. Our use of the PGAS model for opaque distributed data structures lets us hide many of the details of our data structures. Since the processes in a process group are *not* hidden behind an abstract interface, the communicators are at some level exposed to programmers.

Based on the choice of how the communicators are exposed, we consider three alternatives for the design of a distributed GraphBLAS API:

(A) Each job is associated with one and only one communicator at a given time. That is, the communicator is a property of the job. This leaves open the possibility of changing communicators during the execution of a job, but that is not a very common (or particularly useful) operation if the processes available to a job are fixed.

(B) Each GraphBLAS object in the job is associated with one and only one communicator. That is, the communicator is the property of an object. An object is associated with a communicator when it is created. That object can change to a different communicator through a *redistribute* operation, discussed in more detail later. An object must have an associated communicator through its entire lifetime, from creation to deletion.

(C) Each GraphBLAS method call is associated with one and only one communicator. That is, the communicator is the property of a method call. That means that a communicator must be an explicit argument in most, if not all, GraphBLAS methods. For methods such as GrB_free the communicator can be trivially inferred.

We now discuss each of these alternatives in more detail. Even within a given alternative there are several design choices. We will focus on those choices that seem more relevant at this level of the design.

A. One communicator per job

This is the *path of least resistance* to evolve the current GraphBLAS C API into a distributed API. Since there is only one communicator for the job, and it can be static throughout the execution of the job, it suffices to create that communicator at job start. In an MPI distributed environment, it could simply be the default global communicator of MPI, MPI_Comm_world. No changes to the existing GraphBLAS API would be necessary (other than, perhaps, passing the communicator to GrB_init.)

The major drawback of the "one communicator per job" approach is its lack of flexibility in organizing the computation. One cannot split the processes of a job into subgroups, or

control the distribution of data structures. Furthermore, every operation would have to execute on the entire process group which may not always be the more efficient solution.

B. One communicator per object

In this option, the communicator is passed as an argument to object creation methods. This requires a change to the existing GraphBLAS API. Object creation through duplication, however, reuses the communicator so those methods would not change.

APIs of methods that perform GraphBLAS operations (such as GrB_mxm and GrB_mxv) would not need to change under this option. If the objects are local, these operations are strictly local and follow the semantics of the current API. If the objects are distributed, then it is natural to require that all objects share the same communicator. Mixing distributed and local objects may be allowed in some cases. For example, it is reasonable to multiply a distributed matrix by a local vector if the result vector is also local.

If the distributed objects do not all have the same communicator, there are two alternatives. We can provide explicit redistribute methods (*e.g.*, GrB_redistribute), requiring the programmer to first redistribute the inputs so that they all have the same communicator, or perform the redistribution implicitly in the called method.

Redistribution, whether implicit or explicit, would require that every process in both the source and target process-groups must participate in the operation, otherwise the implied collective communication operations would not be possible.

An important benefit of this option is that other than object creation methods, the same code would work with both the sequential and distributed GraphBLAS implementations.

C. One communicator per method

This is the most intrusive option. Almost every GraphBLAS method would need to be augmented with an optional communicator argument. Object creation behaves as in the "one communicator per object" approach above. GraphBLAS operations also explicitly take a communicator, specifying which group of processes should perform the computation.

We would need to define under which circumstances a distributed operation is legal. The easier requirement would be that all the objects passed to the operation use the same communicator as the operation itself. This would make it trivial for the processes in the communicator to cooperate in the execution of the operation. Any difference in communicators would require explicit redistribution.

We can also support automatic redistribution of the objects. Again, for this to work correctly, all tasks in all involved communicators would have to participate in the operation. In this approach, distributed and sequential method calls are clearly different, and all communicators are explicitly identified (and have to be explicitly passed when calling other routines). Code is clear and there is no obfuscation, but at the cost of incompatibility between distributed and (current) sequential APIs.

V. Extending the GraphBLAS context

The needs of multithreaded and distributed memory computing are disjoint suggesting a need for two different APIs. We could merge them, however, by extending the concept of the GraphBLAS context.

The GraphBLAS context is "an instance of the GraphBLAS API implementation as seen by an application". Currently, a program can only have a single GraphBLAS context. We could make GrB_init reentrant so that it could be called multiple times in a single job. Each time it is called a new "instance of the GraphBLAS API implementation" is created. With multiple contexts in a single program, we would need to define a context-handle to pass to GrB_finalize() when terminating a context.

With multiple contexts in a single program, the issues discussed previously for handling communicators in the distributed GraphBLAS would need to be addressed. In particular, we would need to choose between option B (bind the GraphBLAS context to an object when that object is created) or option C (bind the GraphBLAS context to each method when it is called).

With support for multiple GraphBLAS contexts, we would support multithreaded and distributed execution by extending the GrB_Mode parameter passed to call to GrB_init() so it includes:

- Serial semantics: The current GrB_BLOCKING and GrB_NONBLOCKING modes.
- Multithreaded execution: A new descriptor object to which parameters for managing multithreaded execution can be assigned including the number of threads and binding/place-modes needed on NUMA systems.
- Distributed memory execution: A new descriptor object to hold the communicator described earlier. Ideally, the communicator would come from the underlying communication library (such as MPI).

This GraphBLAS context extension will let users intermix GraphBLAS operations executed in a local context, with distributed GraphBLAS operations executed in a distributed context.

Many details need to be worked out. We expect that objects created in one context cannot be used in a different context. We may need to add GraphBLAS methods to convert an object from one context to another. Furthermore, functions called inside a distributed context *must* be called as collective functions among all processes inside the distributed context.

Working out all the changes required to support this expanded scope of the GraphBLAS context will be complicated. We believe these complications would be well justified, however, as GraphBLAS users would benefit greatly from being able to move from serial to multithreaded to distributed execution by changing just a single parameter (the GrB_MODE) within the call to GrB_init()

VI. Related Work

There are few existing distributed sparse matrix libraries with support for semiring algebra. Combinatorial BLAS

(CombBLAS) [6] and Cyclops Tensor Framework (CTF) [11] are perhaps the only two that exist right now. By default, CombBLAS uses a 2D block distribution for both matrices and vectors, though it also includes support for an experimental 3D matrix distribution. By contrast, CTF is more general in terms of its ability to distribute its objects and can handle a large spectrum within a virtual 3D process grid.

Each distributed object in CombBLAS has its own communicator (the "one communicator per object" approach described among the possible options in Section IV). Whenever a CombBLAS function is called on multiple CombBLAS objects, the communicators are compared and only if they are compatible will the computation proceed. CombBLAS executes bulk-synchronously, relying heavily on collective MPI-based inter-process communication using subcommunicators. Furthermore, all CombBLAS functions are efficiently parallelized with OpenMP to take advantage of intra-node parallelism. Since CombBLAS predates GraphBLAS, it does not follow the GraphBLAS API. Furthermore, CombBLAS is missing certain concepts such as accumulators and masks. However, it has the benefit of already having scalable implementations [5] of some of the most challenging GraphBLAS primitives such as GrB_mxm, GrB_extract, and GrB_assign. At Berkeley, we are currently building a distributed GraphBLAS by extending and modifying CombBLAS. We are also planning a more asynchronous GraphBLAS implementation on top of the Berkeley Container Library [4].

Livermore Distributed GraphBLAS (LDGB) is a distributed MPI/C++ GraphBLAS implementation in development at Lawrence Livermore National Laboratory. LDGB uses the API of the GraphBLAS Template Library (GBTL) [1]. While this API does not strictly adhere to the GraphBLAS C API, it provides the same functionality with an interface more familiar to C++ developers. In its current implementation, LDGB uses a global communicator to organize processes. Matrices and vectors are distributed in a 1D cyclic fashion within this global communicator. The GraphBLAS operations are implemented in an asynchronous manner using the YGM communication library [3]. The LDGB project has implemented most of the GraphBLAS 1.2 specification. One notable gap is the GraphBLAS error model, which presents issues with propagating errors in a distributed setting, as also noted by Hughey [8]. LDGB is being developed as a mostly drop-in replacement for GBTL. The project will be made publicly available in the near future.

VII. Conclusion

We face a range of options as we design the distributed GraphBLAS API. Clearly, the API will use a PGAS perspective with GraphBLAS objects as abstract distributed data types. The underlying communication infrastructure will map onto a coordination system such as MPI. This led to 3 options for how communicators will be bound to the GraphBLAS API. We need input from GraphBLAS users to validate which of these options are best.

Another key question is whether we need separate APIs for the multithreaded, serial, and distributed cases. One approach might be to expand the meaning of the graphBLAS context to include multithreaded and distributed execution modes. A programmer could then switch between different execution cases by simply changing the GrB_init() call.

One thing is clear at this point. The design of a distributed memory GraphBLAS specification will be complicated. We need feedback early in this process to make sure the API meets the needs of programmers. It is our hope that this brief position paper will drive such conversations and increase the chances that the 1.0 draft of the distributed GraphBLAS specification will be successful.

Acknowledgments and Disclaimers

We thank the members of the GraphBLAS forum. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM20-0211]. Benjamin Brock and Aydın Buluç were supported in part by the DOE Office of Advanced Scientific Computing Research under contract number DEAC02-05CH11231 and in part by NSF under Award No. 1823034.

References

[1] Graphblas template library (GBTL). https://github.com/cmu-sei/gbtl.

[2] Mohsen Aznaveh, Jinhao Chem, Timothy A. Davis, Balint Hegyi, Scott P. Kolodziej, Timothy G. Mattson, and Gabor Szarnyas. Parallel GraphBLAS with OpenMP. In *SIAM workshop on Combinatorial Scientific Computing*, 2020.

[3] R. Pearce B. Priest, T. Steil, G. Sanders, T. La Fond, and K. Iwabuchi. You've got mail (ygm): Building missing asynchronous communication primitives. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, page 2, May 2019.

[4] Benjamin Brock, Aydın Buluç, and Katherine Yelick. BCL: A cross-platform distributed data structures library. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[5] Aydin Buluç and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

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

[7] Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. The GraphBLAS C API Specification. *GraphBLAS. org, Tech. Rep.*, version 1.3.0, 2019.

[8] Curtis Hughey. Tumbling down the GraphBLAS rabbit hole with SHMEM. In *Workshop on OpenSHMEM and Related Technologies*, pages 125–136. Springer, 2018.

[9] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[10] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[11] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *IEEE Intr. Symposium on Parallel & Distributed Processing*, pages 813–824, 2013.