SPARSE Project Report

Andrea Pelosi

In this document we present *SPARSE* project and sparse.jl, its associated Julia package developed as part of the course IN480 - Parallel and Distributed Computing, Roma Tre University. In *SPARSE*, we propose and benchmark against each other three different Breadth First Search (*BFS*) algorithm implementations. Two of them are built upon GraphBLAS, a mathematical abstraction to represent and operate with graphs using just a narrow set of linear algebra primitives, while the other one relies upon Julia standard packages to operate with sparse matrices (that could be used to represent graphs).

SPARSE source code can be found at *sparse.jl*.

Preliminaries

Why GraphBLAS?

Contemporary computer architectures are good at processing linear and hierarchical data structures, such as lists, stack or trees. They do not behave equally well with graphs. Graph algorithms are indeed challenging to program: issues such as poor locality, frequent cache misses and difficulty of parallelization demolish graph algorithms performance, and unfortunately optimizations reduce portability. In order to address these problems, a group of graph algorithms researchers came up with GraphBLAS.

GraphBLAS aims to set standard building blocks for graph representation and computation in the language of linear algebra. Graph algorithms were already expressed in terms of operations on *sparse* matrices (matrices in which the majority of the elements are zero) but GraphBLAS goes a step further trying to standardize the approach. Amongst the many improvements GraphBLAS could lead, we point out efficiency, portability, compactness, composability and flexibility.

To appreciate these advantages, we first introduce the mathematics upon which GraphBLAS is built. Notice though that the following introduction is far from comprehensive: we describe only what is relevant to fully understand *SPARSE*, along with some additional examples.

Theoretical foundations

Matrices as graph displaying tools

A graph G is a pair $G = (V, E)$ where V and E are sets whose elements are called vertices and edges respectively. Using matrices, it is possible to display graphs in different ways. One of the most common matrix representations for graphs is the *adiacency matrix*.

Adiacency matrix Given a graph $G = (V, E)$ where $|V| = n$, its *adiacency matrix* A is a matrix with n rows and n columns where $A(i, j) = 1$ if there is an edge going from the vertex i to the vertex j and $A(i, j) = 0$ otherwise. If G is a *weighted graph* (namely a graph in which every edge has weight), $A(i, j) = w$ if there is an edge going from i to j that weights w.

$A(i, j)$ denotes the element in the ith row and the jth column.

The canonical GraphBLAS matrix has m rows and n columns and is defined by the following mapping:

$$\mathbf{A} : I \times J \to \mathbb{S}$$

where $I, J \subseteq \mathbb{Z}$ sets of indices with m and n elements respectively and $\mathbb{S}$ a set of scalars, with $\mathbb{S} \in \{\mathbb{Z}, \mathbb{R}, \mathbb{C}, \dots\}$.

Thus, without loss of generality, we can denote a GraphBLAS matrix as:

$$\mathbf{A} : \mathbb{S}^{m \times n}$$

A GraphBLAS column vector is a GraphBLAS matrix where $n = 1$; a GraphBLAS row vector is a GraphBLAS matrix where $m = 1$. For the sake of brevity we shall refer to matrices and vectors instead of GraphBLAS matrices and GraphBLAS vectors. Notice that, even though vectors and matrices are usually sparse (and, for instance, *SPARSE* is one of such cases), the correctness of the GraphBLAS model is independent from sparsity.

GraphBLAS fundamental building block

The most important GraphBLAS operation is matrix-matrix multiplication, since it allows to build a large variety of graph algorithms. Traditional matrix-matrix multiplication requires the use of arithmetic sum and product. Using different kinds of operators and different domains while working with matrices, allows to further extend the variety of graph operations that could be written. This extended matrix-matrix multiplication is performed on an algebraic *semiring*:

Semiring Given a set D and two binary operations $\oplus$ and $\otimes$ (called addition and multiplication respectively), a *semiring* is a triple $(D, \oplus, \otimes)$ where:

- $(D, \oplus)$ is a commutative monoid and 0 is the additive identity element;
- $(D, \otimes)$ is a monoid and 1 is the multiplicative identity element;
- Multiplication distributes over addition;

- The element 0 is a multiplicative annihilator.

One of the main advantages GraphBLAS provides is that $\oplus$ and $\otimes$ could be user-defined. Here there are some semirings examples widely used with graph algorithms (from Buluç et al. (2017a)):

Semiring	$\oplus$	$\otimes$	domain	**0**	**1**
Standard arithmetic	$+$	$\times$	$\mathbb{R}$	0	1
max-plus algebras	max	$+$	$\{-\infty \bigcup \mathbb{R}\}$	$-\infty$	0
Galois fields	xor	and	$\{0, 1\}$	0	1
Power set algebras	$\bigcup$	$\bigcap$	$\mathscr{P}(\mathbb{Z})$	$\emptyset$	U

Most common operations overview

Besides matrix-matrix multiplication, there are a few other fundamental GraphBLAS operations by which a lot of graph algorithms can be expressed. Not all of them can be found in sparse.jl source code. However, we present them, along with a brief explanation, to give some insight of what can be done with GraphBLAS. Note that $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices and $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are vectors over the semiring $(\mathbb{S}, \oplus, \otimes)$. For a complete list, refer to Kepner (2017) and Buluç et al. (2017b).

matrix build: Build a sparse matrix from row, column and value tuples.
vector build: Build a sparse vector from index value tuples.
assign: $\mathbf{C}(\mathbf{i}, \mathbf{j}) \oplus = \mathbf{A}(\mathbf{i}, \mathbf{j})$. Perform an assignment of the values at $\mathbf{A}(\mathbf{i}, \mathbf{j})$ to the corresponding vertices in $\mathbf{C}$.
mxm: $\mathbf{C} = \mathbf{A} \oplus . \otimes \mathbf{B}$. Perform matrix-matrix multiplication between $\mathbf{A}$ and $\mathbf{B}$ and store the result in the matrix $\mathbf{C}$.
vxm: $\mathbf{w}^T = \mathbf{v}^T \oplus . \otimes \mathbf{A}$. Perform vector-matrix multiplication between $\mathbf{v}^T$ and $\mathbf{A}$ and store the result in the vector $\mathbf{w}^T$.
mxv: $\mathbf{w} = \mathbf{A} \oplus . \otimes \mathbf{v}$. Perform matrix-vector multiplication between $\mathbf{A}$ and $\mathbf{v}$ and store the result in the vector $\mathbf{w}$.
eWiseAdd: $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{w} = \mathbf{u} \oplus \mathbf{v}$. Perform element-wise $\oplus$ between $\mathbf{A}$ and $\mathbf{B}$ or $\mathbf{u}$ and $\mathbf{v}$, respectively. Store the result in the matrix $\mathbf{C}$ or the vector $\mathbf{w}$, respectively.
eWiseMult: $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ or $\mathbf{w} = \mathbf{u} \oplus \mathbf{v}$. Perform element-wise $\otimes$ between $\mathbf{A}$ and $\mathbf{B}$ or $\mathbf{u}$ and $\mathbf{v}$, respectively. Store the result in the matrix $\mathbf{C}$ or the vector $\mathbf{w}$, respectively.
apply: $\mathbf{C} \oplus = f(\mathbf{A})$. Apply the unary function f to the matrix $\mathbf{A}$. Store the result in the matrix $\mathbf{C}$.

Main packages used

SuiteSparse:GraphBLAS.jl

Two different GraphBLAS implementations written in Julia are used to build *BFS* algorithm:

- SuiteSparseGraphBLAS.jl by Abhinav Mehndiratta. It tries to adhere as much as possible to the GraphBLAS C API Specification (see https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf), and has currently implemented the majority of fundamental GraphBLAS primitives;
- SuiteSparseGraphBLAS.jl by Computational Visual Design Lab, laboratory located in RomaTre University, related to the Department of Computer Engineering and the Department of Mathematics and Physics (for additional information, visit http://cvdlab.org/). This implementation is a fork of the former, and brings fundamental GraphBLAS primitive in a more "Julian" fashion.

Notice that, precisely speaking, both the first and the second implementations presented above are just wrappers for the C library SuiteSparse:GraphBLAS (see https://github.com/DrTimothyAldenDavis/GraphBLAS), one of the main full implementation of GraphBLAS standard, that implements in turn the GraphBLAS C API Specification.

The main reason why we use different GraphBLAS wrappers is to show how two slightly different implementations of the same library (difference not easily noticeable from a user point of view) could influence ease of use and investigate if they influence algorithm final performance, too.

SparseArrays.jl

The last *BFS* implementation is based on the standard library package SparseArrays.jl and shares the same core idea as the other two in a way that will be explicited in the next section.

Algorithm description

The (undirected) graph which we want to test is represented as the $n \times n$ adiacency matrix A. Algorithm is given the source s from which *BFS* starts, too. The core idea behind the different implementations is the same:

there are two vectors v and q of length n, representing respectively the level of each graph node in the *BFS* and the set of nodes belonging to the *frontier* (namely the nodes discovered in one of the at most n iterations that *BFS* does). At each *BFS* iteration, there is a matrix-vector product between A and q resulting in a frontier update (one can easily verify that; for an explicit example of this property, see Kepner (2017)). Vector v is updated: $v[i] = k$ if the node i is

visited (namely, appears in the frontier) for the first time at the k-th iteration. Nodes already visited are masked out from the frontier, and the algorithm loops until n iterations are done or until the frontier empties.

GraphBLAS based *BFSs* have several advantages over the SparseArrays based *BFS*: just a few standard GraphBLAS operations are needed in order to write the algorithm and, in the algorithm design phase, it is easy to choose the semiring that best suits the wanted operation (notice that, just choosing another semiring, one could go a step further and possibly analyze different properties of the same graph with very little effort). Moreover, there is no need to build from scratch a heavily optimized algorithm, since every GraphBLAS operation is internally optimized; thus, using different GraphBLAS primitives results in an efficient algorithm.

On the contrary, building a SparseArrays based *BFS* (without referring to one of the highly efficient *BFS* implementations present in literature) requires explicit optimizations and clever usage of the features Julia provides, and even that is not enough to produce something which could reach the performance of GraphBLAS based *BFS*.

Benchmark results

We present benchmark results for the three *BFS* algorithms implemented in sparse.jl. In order to have a statistically accurate benchmark, we use BenchmarkTools.jl, a Julia package devoted to track Julia code performance in a statistically accurate manner. All the tests were executed on a machine with a Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz capable of $4.594591126143963e10$ *FLOPs* (Floating Point Operations per second), estimated through the Julia function `peakflops()`.

`GraphBLAS_bfs!`

`GraphBLAS_bfs!` with a 100×100 input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  432 bytes
  allocs estimate:  23
  --------------
  minimum time:     29.228 μs (0.00% GC)
  median time:      30.040 μs (0.00% GC)
  mean time:        31.441 μs (0.00% GC)
  maximum time:     241.677 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

`GraphBLAS_bfs!` with a 1000×1000 input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  432 bytes
  allocs estimate:  23
  --------------
  minimum time:     33.464 μs (0.00% GC)
  median time:      34.243 μs (0.00% GC)
  mean time:        35.793 μs (0.00% GC)
  maximum time:     275.443 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

`GraphBLAS_bfs!` with a $10^5 \times 10^5$ input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  576 bytes
  allocs estimate:  32
  --------------
  minimum time:     364.527 μs (0.00% GC)
  median time:      372.448 μs (0.00% GC)
  mean time:        379.128 μs (0.00% GC)
  maximum time:     1.795 ms (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

`GraphBLAS_bfs!` with a $10^7 \times 10^7$ input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  1.41 KiB
  allocs estimate:  86
  --------------
  minimum time:     60.763 ms (0.00% GC)
  median time:      60.934 ms (0.00% GC)
  mean time:        61.247 ms (0.00% GC)
  maximum time:     63.149 ms (0.00% GC)
  --------------
  samples:          82
  evals/sample:     1
```

GraphBLAS_bfs_cvd

`GraphBLAS_bfs_cvd` with a 100×100 input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  9.98 KiB
  allocs estimate:  211
```

```
  --------------
  minimum time:     257.366 μs (0.00% GC)
  median time:      263.248 μs (0.00% GC)
  mean time:        267.457 μs (0.40% GC)
  maximum time:     12.179 ms (43.83% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

`GraphBLAS_bfs_cvd` with a 1000×1000 input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  28.91 KiB
  allocs estimate:  606
  --------------
  minimum time:     738.113 μs (0.00% GC)
  median time:      746.118 μs (0.00% GC)
  mean time:        755.463 μs (0.36% GC)
  maximum time:     7.536 ms (58.43% GC)
  --------------
  samples:          6615
  evals/sample:     1
```

`GraphBLAS_bfs_cvd` with a $10^5 \times 10^5$ input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  10.06 KiB
  allocs estimate:  216
  --------------
  minimum time:     264.206 μs (0.00% GC)
  median time:      271.039 μs (0.00% GC)
  mean time:        280.009 μs (0.38% GC)
  maximum time:     12.034 ms (44.44% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

`GraphBLAS_bfs_cvd` with a $10^7 \times 10^7$ input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  10.06 KiB
  allocs estimate:  216
  --------------
  minimum time:     295.811 μs (0.00% GC)
  median time:      305.381 μs (0.00% GC)
  mean time:        310.034 μs (0.34% GC)
  maximum time:     12.375 ms (43.66% GC)
```

```
  --------------
  samples:          10000
  evals/sample:     1
```

bfs_primitive

`bfs_primitive` with a 100×100 input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  2.30 KiB
  allocs estimate:  10
  --------------
  minimum time:     68.504 μs (0.00% GC)
  median time:      208.141 μs (0.00% GC)
  mean time:        277.041 μs (0.07% GC)
  maximum time:     2.508 ms (79.66% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

`bfs_primitive` with a 1000×1000 input matrix:

```
BenchmarkTools.Trial:
  memory estimate:  45.72 KiB
  allocs estimate:  21
  --------------
  minimum time:     18.678 ms (0.00% GC)
  median time:      31.282 ms (0.00% GC)
  mean time:        800.673 ms (0.00% GC)
  maximum time:     6.207 s (0.00% GC)
  --------------
  samples:          8
  evals/sample:     1
```

A few consideration emerge from the benchmark. While `GraphBLAS_bfs!` algorithm is a little faster for small inputs compared with `GraphBLAS_bfs_cvd`, the latter becomes faster for very large inputs. On the other side, the former has lower memory usage even for large inputs compared with the latter. As regards to `bfs_primitive` algorithm, we observe that it is not as efficient as the other two, to the point that running `bfs_primitive` on large inputs becomes impractical (especially for the amount of time needed by the computation to give an output).

Conclusions

In this document we have talked about *SPARSE* project and its associated Julia package sparse.jl. Our aim was to present GraphBLAS, a cutting edge

mathematical abstraction for operating with very large graphs using the language of linear algebra, and to benchmark three different *BFS* implementations in order to show the advantages that working with a high performance standard such as GraphBLAS could bring to the world of graph algorithms.

References

Buluç, Aydin, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017a. "Design of the GraphBLAS API for c." In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 643–52. IEEE.

Buluç, Aydin, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017b. "The Graphblas c Api Specification." *GraphBLAS. Org, Tech. Rep.*

Kepner, Jeremy. 2017. "GraphBLAS Mathematics-Provisional Release 1.0." *GraphBLAS. Org, Tech. Rep.*