Push-Pull on Graphs is Column- and Row-based SpMV Plus Masks

Carl Yang
University of California, Davis
Lawrence Berkeley National Laboratory
ctcyang@ucdavis.edu

Aydın Buluç
Lawrence Berkeley National Laboratory
abuluc@lbl.gov

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

Push-pull, also known as direction-optimized breadth-first-search (DOBFS), is a key optimization for making breadth-first-search (BFS) run efficiently. Linear algebra-based frameworks have advantages in conciseness, performance and portability. However, there is no work in literature describing how to implement it within a linear algebra-based framework. Our work shows that DOBFS fits well within the linear algebra-based framework.

KEYWORDS

sparse matrix multiplication, breadth-first search, graph algorithms

1 INTRODUCTION

Push-pull, also known as direction-optimized breadth-first-search (DOBFS), is a key optimization for making breadth-first-search (BFS) run efficiently [3]. According to the Graph Algorithm Platform [2], no fewer than 32 out of the top 37 entries on the Graph500 benchmark (a suite for ranking the fastest graph frameworks in the world) use direction-optimizing BFS. Since its discovery, it has been extended to other traversal-based algorithms [5, 11]. One of our contributions in this paper is factoring Beamer's direction-optimized BFS into 3 separable optimizations, and analyzing them independently—both theoretically and empirically—to determine their contribution to the overall speed-up. This allows us to generalize these optimizations to other graph algorithms, as well as fit it neatly into a linear algebra-based graph framework. These 3 optimizations are, in increasing order of specificity:

(1) Change of direction: Use the *push* direction to take advantage of knowledge that the frontier is small, which we term *input sparsity*. When the frontier becomes large, go back to the *pull* direction.
(2) Masking: In the *pull* direction, there is an asymptotic speed-up if we know *a priori* the subset of vertices to be updated, which we term *output sparsity*.
(3) Early-exit: In the *pull* direction, once a single parent has been found, the computation for that undiscovered node ought to exit early from the search.

ICPP '18, Eugene, Oregon

DOI: 0000/0000

GraphBLAS is an effort by the graph analytics community to formulate graph algorithms as sparse linear algebra [6]. The goal of the GraphBLAS API specification is to outline the common, high-level operations such as vector-vector inner product, matrix-vector product, matrix-matrix product, and define the standard interface for scientists to use these functions in a hardware-agnostic manner. This way, the runtime of the GraphBLAS implementation can make the difficult decisions about optimizing each of the GraphBLAS operations on a given piece of hardware.

Previous work by Beamer et al. [4] and Besta et al. [5] have observed that push and pull correspond to column- and row-based matvec (Optimization 1). However, this realization has not made it into the sole GraphBLAS implementation in existence so far, namely SuiteSparse GraphBLAS [9]. In SuiteSparse GraphBLAS, the BFS executes in only the forward (push) direction.

The key distinction between our work and that of Shun and Blelloch [11], Besta et al. [5], and Beamer et al. [4] is that while they take advantage of *input sparsity* using change of direction (Optimization 1), they do not analyze using *output sparsity* through masking (Optimization 2), which we show theoretically and empirically is critical for high performance. Furthermore, we submit this speed-up extends to all algorithms for which there is *a priori* information regarding the sparsity pattern of the output such as triangle counting [1], adaptive PageRank [10], batched betweenness centrality [6], and maximal independent set [7].

Since the input vector can be either sparse or dense, we refrain from referring to this operation as SpMSpV (sparse matrix-sparse vector) or SpMV (sparse matrix-dense vector). Instead, we will refer to it as matvec (short for matrix-vector multiplication and known in GraphBLAS as GrB_mxv).

2 APPLICATIONS

The details of these optimizations are in the full paper of this conference [12], so we will give some details here on how masking can be used in the four applications mentioned above.

2.1 Triangle counting

Algorithm 1 gives the algorithm for triangle counting as described by Azad, Buluç and Gilbert [1].

2.2 Adaptive PageRank

Algorithm 2 gives the algorithm for adaptive PageRank as described by Kamwar, Haveliwala and Golub [10].

Algorithm 1 Triangle counting with **A** as the mask.

```
procedure TriangleCount(Undirected graph A)
    L ← lower_triangular(A)                ▷ tril() in MATLAB
    U ← upper_triangular(A)                ▷ triu() in MATLAB
    C ← A .∗ (L × U)
    num_tri ← nnz(C)
return num_tri
end procedure
```

Algorithm 2 Adaptive PageRank with **v** as the mask.

```
procedure AdaptivePR(Directed graph A, Initial PageRank x^(0), Absolute tolerance ϵ)
    v ← {1, 1, …, 1}
    k ← 0
    while δ < ϵ do
        x^(k+1) ← v .∗ (A × x^(k))
        v ← |x^(k+1) − x^(k)| < ϵ
        k ← k + 1
        δ = ‖Ax^(k) − x^(k)‖_1
    end while
return x^k
end procedure
```

2.3 Batched betweenness centrality

Algorithm 3 gives the algorithm for batched betweenness centrality as described by Buluç et al. [6].

2.4 Maximal independent set

Algorithm 4 gives an algorithm for maximal independent set as described by Buluç et al. [7].

3 CONCLUSION

In our full paper [12], we have demonstrated that push-pull corresponds to the concept of column- and row-based masked matvec. We have experimental evidence that masking is advantageous from both a theoretical and empirical standpoint. In this paper, we have demonstrated that masking is a very common operation in graph analytics. It is applicable to applications such as triangle counting, adaptive PageRank, batched betweenness centrality, and maximal independent set. A possible future research direction would be to provide experimental evidence for these four algorithms against existing, state-of-the-art implementations.

Algorithm 3 Batched betweenness centrality with mask operations in Lines 17, 22 and 23.

```
1:  procedure BatchedBC(Directed graph A, Number of vertices n, Batch size b, Vector of sources s)
2:      σ ← n × n × b tensor initialized to 0
3:      V ← n × b matrix initialized to 0
4:      for i = 0, 1, 2, …, b do
5:          V(i, s(i)) ← 1
6:      end for
7:      U ← n × b matrix initialized to 1
8:      W ← n × b matrix initialized to 0
9:      δ ← n × 1 vector initialized to −b
10:     F ← V
11:     N ← 1/V                              ▷ elementwise operation
12:     d ← 0
13:     c ← 1
14:     while c > 0 do
15:         σ(d, :, :) ← F
16:         V ← V + F
17:         F ← ¬V .∗ (A^T × F)
18:         c ← nnz(F)
19:         d ← d + 1
20:     end while
21:     while d ≥ 2 do
22:         W ← σ(d, :, :) .∗ (U .∗ N)
23:         W ← σ(d − 1, :, :) .∗ (A × W)
24:         U ← U + (W .∗ V)
25:         d ← d − 1
26:     end while
27:     δ ← δ + Σ_{j=1}^{b} U(:, j)          ▷ reduction across U columns
28: return δ
29: end procedure
```

Algorithm 4 Maximal independent set with **A** as the mask.

```
procedure MaximalIndependentSet(Undirected graph A, Number of rows n, Hash function hash())
    i ← {0, 0, …, 0}                ▷ initialize independent set to false
    c ← 1
    while c > 0 do
        for i = 0, 1, 2, …, n and i ∈ c do
            f(i) ← hash(i)
        end for
        m ← c .∗ (A × f)            ▷ Using max-times semiring
        v ← f ≥ m
        i ← i + v                   ▷ using Boolean OR semiring
        c ← ¬v .∗ c
        c ← nnz(c)
    end while
return i
end procedure
```

REFERENCES

[1] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *IPDPSW*. IEEE, 804–811.

[2] Scott Beamer. [n. d.]. Direction-Optimizing Breadth-First Search. http://gap.cs.berkeley.edu/dobfs.html. ([n. d.]). Accessed: 2018-01-18.

[3] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-Optimizing Breadth-First Search *(SC)*. IEEE, Article 12, 10 pages.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing Pagerank Communication via Propagation Blocking *(IPDPS)*. IEEE, 820–831.

[5] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations *(HPDC)*. ACM, 93–104.

[6] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C *(IPDPSW)*. IEEE, 643–652.

[7] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. 2017. *The GraphBLAS C API Specification*. Rev. 1.1.

[8] S. Changpinyo, M. Sandler, and A. Zhmoginov. 2017. The power of sparsity in convolutional neural networks. *arXiv preprint arXiv:1702.06257* (2017).

[9] Tim Davis. 2017. suitesparse : a suite of sparse matrix software. http://faculty.cse.tamu.edu/davis/suitesparse.html. (2017). Accessed: 2018-01-18.

[10] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. 2004. Adaptive methods for the computation of PageRank. *Linear Algebra Appl.* 386 (2004), 51–65.

[11] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory *(PPoPP '13)*. ACM, New York, NY, USA, 135–146.

[12] Carl Yang, Aydin Buluc, and John D Owens. 2018. Implementing Push-Pull Efficiently in GraphBLAS. *ICPP* (2018).