To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations

Maciej Besta¹, Michal Podstawski² ³, Linus Groner¹, Edgar Solomonik⁴, Torsten Hoefler¹
¹ Department of Computer Science, ETH Zurich; ² Perform Group Katowice.; ³ Katowice Institute of Information Technologies; ⁴ Department of Computer Science, University of Illinois at Urbana-Champaign
maciej.besta@inf.ethz.ch, michal.podstawski@performgroup.com, gronerl@student.ethz.ch, solomon2@illinois.edu, htor@inf.ethz.ch

ABSTRACT

We reduce the cost of communication and synchronization in graph processing by analyzing the fastest way to process graphs: pushing the updates to a shared state or pulling the updates to a private state. We investigate the applicability of this push-pull dichotomy to various algorithms and its impact on complexity, performance, and the amount of used locks, atomics, and reads/writes. We consider 11 graph algorithms, 3 programming models, 2 graph abstractions, and various families of graphs. The conducted analysis illustrates surprising differences between push and pull variants of different algorithms in performance, speed of convergence, and code complexity; the insights are backed up by performance data from hardware counters. We use these findings to illustrate which variant is faster for each algorithm and to develop generic strategies that enable even higher speedups. Our insights can be used to accelerate graph processing engines or libraries on both massively-parallel shared-memory machines as well as distributed-memory systems.

Site: https://spcl.inf.ethz.ch/Research/Parallel_Programming/PushPull

1 INTRODUCTION

Graph processing underlies many computational problems in social network analysis, machine learning, computational science, and others [33]. Designing efficient parallel graph algorithms is challenging due to several properties of graph computations such as irregular communication patterns or little locality. These properties lead to expensive synchronization and movements of large data amounts on shared- and distributed-memory (SM, DM) systems.

Direction optimization in breadth-first search (BFS) [4] is one of the mechanisms that are used to alleviate these issues. It combines the traditional top-down BFS (where vertices in the active frontier iterate over all unvisited neighbors) with a bottom-up scheme (where unvisited vertices search for a neighboring vertex in the active frontier [48]). Combining these two approaches accelerates BFS by ≈2.4x on real-world graphs such as citation networks [4].

We first illustrate that distinguishing between bottom-up and top-down BFS can be generalized to many other graph algorithms, where updates can be either pushed by a thread to the shared state (as in the top-down BFS), or pulled to a thread’s private state (as in the bottom-up BFS). As another example, consider a PageRank (PR) computation and assume a thread X is responsible for a vertex v. X can either push v’s rank to update v’s neighbors, or it can pull the ranks of v’s neighbors to update v [52]. Despite many differences between PR and BFS (e.g., PR is not a traversal), PR can similarly be viewed in the push-pull dichotomy.

This notion sparks various questions. Can pushing and pulling be applied to any graph algorithm? How to design push and pull variants of various algorithms? Is pushing or pulling faster? When and why? Does it depend on the utilized programming model and abstraction? When and how can pushing or pulling be accelerated?

We seek to answer these and other questions and provide the first extensive analysis on the push-pull dichotomy in graph processing. Now, this dichotomy was identified for some algorithms [4, 52] and was used in several graph processing frameworks, such as Ligra [46] and Gemini [57]. Yet, none of these works analyzes the differences in formulations, complexity, and performance between the two approaches for various algorithms, environments, or models.

As a motivation, consider Figure 1 with the results of our push/pull variants of graph coloring [6]. They unveil consistent advantages of pushing. The figure also shows the speedup from a strategy GrS (“Greedy-Switch”) that (1) reduces the number of memory access with a traversal-based graph coloring, and (2) switches between push- or pull-based scheme and an optimized greedy variant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4699-3/17/06. . . $15.00.
https://doi.org/10.1145/3078597.3078616

We provide the following contributions:

- We apply the push-pull dichotomy to various classes of graph algorithms and obtain detailed formulations of centrality schemes, traversals, calculating minimum spanning trees, graph coloring, and triangle counting. We also show that several existing graph processing schemes are included in the push-pull dichotomy.
- We analyze pushing and pulling with PRAM and derive the differences in the amount of synchronization and communication in both variants of the considered algorithms.
- We analyze performance of push- and pull-based algorithms for both SM and DM systems that represent fat-memory nodes and supercomputers. Various programming models are incorporated, including threading, Message Passing (MP), and Remote Memory Access (RMA) [20] for various classes of graphs. For detailed insights, we gather performance data (e.g., cache misses or issues branches and atomic instructions) using PAPI counters.
We first describe the necessary concepts.

2 MODELS, NOTATION, CONCEPTS

2.1 Machine Model and Simulations

Parallel Random Access Machine (PRAM) [17] is a well-known model of a parallel computer. There are \( P \) processors that exchange data by accessing cells of a shared memory of size \( M \) cells. They proceed in tightly-synchronized steps: no processor executes an instruction \( i + 1 \) before all processors complete an instruction \( i \). An instruction can be a local computation or a read/write from/to the memory. We use \( S \) and \( W \) to denote \textit{time} and \textit{work}: the longest execution path and the total instruction count. There are three PRAM variants with different rules for concurrent memory accesses to the same cell. EREW prevents any concurrent accesses. CREW allows for concurrent reads but only one write at a time. CRCW enables any concurrent combination of reads/writes and it comes with multiple flavors that differently treat concurrent writes. We use the Combining CRCW (CRCW-CB) [25]: the value stored is an associative and commutative combination of the written values.

Now, a simulation of one PRAM machine on another is a scheme that enables any instruction from the former to be executed on the latter. Simulation schemes are useful when one wants to port an algorithm developed for a stronger model that is more convenient for designing algorithms (e.g., CRCW) to a weaker one that models hardware more realistically (e.g., CREW). The used simulations are:

Simulating CRCW/CREW on CREW/EREW Any CRCW with \( M \) cells can be simulated on an \( MP\)-cell CREW/EREW with a slowdown of \( \Theta(\log n) \) and memory \( MP \) (similarly to simulating a CREW on an EREW) [25].

Limiting \( P \) (LP) A problem solvable on a \( P\)-processor PRAM in \( S \) time can be solved on a \( P'\)-processor PRAM \( (P' \prec P) \) in time \( S' = \left\lceil \frac{S}{P'} \right\rceil \) for a fixed memory size \( M \).

2.2 Graph Model, Layout, and Notation

A tuple \((V, E)\) models an undirected graph \( G; V \) is a set of vertices and \( E \subseteq V \times V \) is a set of edges; \(|V| = n \) and \(|E| = m\). \( d(v) \) and \( N(v) \) are the degree and the neighbors of a vertex \( v \). (The non-negative) weight of an edge \((v, w)\) is \( w_{(v, w)} \). We denote the maximum degree for a given \( G \) as \( d \), \( d_{in} \) (in-degree), and \( d_{out} \) (out-degree). The average degree is denoted with a bar \( \bar{d} \). \( G \)’s diameter is \( D \).

The neighbors of each \( v \) form an array. The arrays of all the vertices form a contiguous array accessed by all the threads; we also store offsets into the array that determine the beginning of the array of each vertex. The whole representation takes \( n + 2m \) cells.

We partition \( G \) by vertices (1D decomposition) [11]. We denote the number of used threads/processes as \( P \). We name a thread (process) that owns a given vertex \( v \) as \( t(v) \). We focus on label-setting algorithms. In some of the considered schemes (e.g., PageRank) the number of iterations \( L \) is a user-specified parameter.

2.3 Atomic Operations

Atomic operations (atomics) appear to the system as if they occur instantaneously. They are used in lock-free graph computations to perform fine-grained updates [24, 39]. Here, we use CPU atomics that operate on integers. We now present the relevant operations:

- \textit{Fetch-and-Add(\texttt{target}, \texttt{arg}) (FAA)}: it increases *\texttt{target} by \texttt{arg} and also returns *\texttt{target}’s previous value.
- \textit{Compare-and-Swap(\texttt{target}, \texttt{compare}, \texttt{value}, \texttt{result}) (CAS)}: if *\texttt{target} == \texttt{compare} then *\texttt{target} = \texttt{value} and *\texttt{result} = true are set, otherwise *\texttt{target} is not changed and *\texttt{result} = false.

2.4 Communication & Synchronization

Unless stated otherwise, we associate \textit{communication} with: intra- or inter-node reads and writes, messages, and collective operations other than barriers. \textit{Synchronization} will indicate: any atomic operations, locks, and any form of barrier synchronization.

3 PUSH-PULL: APPLICABILITY

We first analyze what algorithms can be expressed in the push-pull (PP) dichotomy; we revisit existing schemes and discuss new cases.

3.1 PageRank (PR)

PR [10] is an iterative centrality algorithm that obtains the \textit{rank} of each vertex \( v: r(v) = (1 - f)/|V| + \sum_{v \in N(v)} (f \cdot r(w)/d(w)); f \) is the \textit{damp factor} [10]. PR is used to rank websites.

Pushing and Pulling? PR can be expressed in both [52]. In the former, \( t[v] \) updates all \( v \)’s neighbors with a value \( r(v)/d(v) \) (it pushes the value from \( v \) to \( N(v) \)). In the latter, \( t[v] \) updates \( v \) with values \( r(u)/d(u), u \in N(v) \) (it pulls the updates from \( N(v) \) to \( v \)).

3.2 Triangle Counting (TC)

In TC, one counts the number of triangles that each vertex \( v \in V \) is a part of; a triangle occurs if there exist edges \( \{v, w\}, \{w, u\}, \{v, u\} \) where \( u, w \in V \) and \( u, w \neq v, u \neq w \). TC is used in various statistics and machine learning schemes [43] and libraries such as igraph [14].

Pushing and Pulling? This algorithm is also expressible in both schemes. Consider a thread \( t[v] \) that counts the number of triangles associated with a vertex \( v \) \( t(c)(v) \). It iterates over \( N(v) \) and, for each \( u \in N(v) \), it iterates over \( N(u) \) and checks if \( \exists w \in V, v \neq w \neq u \) such that \( w \in N(u) \cap N(v) \); the final sums are divided by 2 at the end. If yes, then, in the push variant, it increments either one of \( t(c)(u) \) and \( t(c)(w) \) while in the pull scheme it increments \( t(c)(v) \).

3.3 Breadth-First Search (BFS)

The goal of BFS [13] is to visit each vertex in \( G \). The algorithm starts with a specified \textit{root} vertex \( r \) and visits all its neighbors \( N(r) \). Then, it visits all the unvisited neighbors of the root’s neighbors, and continues to process each level of neighbors in one step. BFS represents graph traversals and is used the HPC benchmark Graph500 [39].

Pushing and Pulling? There exist both variants. The former is the traditional \textit{top-down} BFS where \( t[v] \) (if \( v \) is in a frontier) checks each unvisited \( u \in N(v) \) and adds it to the next frontier \( F \) (it pushes the updates from \( v \) to \( N(v) \)). The latter is the \textit{bottom-up} approach [4, 48]: in each iteration every unvisited vertex \( u \) is tested if it has a parent in \( F \) (the updates are pulled from \( N(u) \) to \( u \)).

3.4 Single Source Shortest Path (SSSP)

SSSP outputs the distance from a selected source vertex \( s \) to all other vertices. We consider \( \Delta \)-Stepping (SSSP-\( \Delta \)) [37] that combines
the well-known Dijkstra’s and Bellman-Ford algorithms by trading work-optimality for more parallelism. It groups vertices into buckets and only vertices in one bucket can be processed in parallel. SSSP has applications in, e.g., operations research.

**Pushing and Pulling?** Both are applicable when relaxing edges of each vertex \( v \) from the current bucket. In the former, \( v \) pushes relaxation requests to its neighbors in the buckets with unsettled vertices. In the latter, vertices in unsettled buckets look for their neighbors in the current bucket and perform (pull) relaxations. A similar scheme was used in the DM implementation of SSSP-\(\Delta \) [12].

### 3.5 Betweenness Centrality (BC)

BC measures the importance of a vertex \( v \) based on the number of shortest paths that lead through \( v \). Let \( \sigma_0(v) \) be the number of shortest paths between two vertices \( s, t \), and let \( \sigma_{\gamma}(v) \) be the number of such paths that lead through \( v \). BC of \( v \) equals \( bc(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{\gamma}(v)}{\sigma_0(v)} \).

Here, we consider Brandes’ algorithm [9, 41]. Define the dependency of a source vertex \( s \) on \( v \) as: \( \delta(s,v) = \sum_{t \in V} \frac{\sigma_{\gamma}(v)}{\sigma_0(v)} \) where \( \delta(s,v) \) satisfies the following recurrence: \( \delta(s,v) = \sum_{w: v \neq w \in V} \frac{\sigma_{\gamma}(v)}{\sigma_0(v)}(1 + \delta(s,w)) \); \( \delta(s,w) \) is a list of immediate predecessors of \( w \) in the shortest paths from \( s \) to \( w \). Brandes’ scheme uses this recurrence to compute \( bc(v) \) in two phases. First, BFS or SSSP traversals compute \( \delta(s,v) \) and \( \sigma_{\gamma}(v) \), \( \forall s,v \in V \), obtaining a tree \( T \) over \( G \). Next, \( T \) is traversed backwards (from the highest to the lowest distance) to compute \( \delta(s,v) \) and \( bc(v) \) based on the equations above. BC is a complex centrality scheme used in biology, transportation, and terrorism prevention [3].

**Pushing and Pulling?** Both parts of Brandes BC can be expressed using push and pull. The first phase can compute shortest path information using either top-down or bottom-up BFS or push-and-pull-based versions of SSSP. The second phase (backward accumulation) may also be cast as BFS from a starting front. In particular, one can either push partial centrality scores to predecessors or pull them from lists of successors [34].

### 3.6 Graph Coloring (GC)

GC assigns colors to vertices so that no two incident vertices share the same color and the number of colors is minimized. We consider Boman graph coloring (BGC) [6]. Here, each iteration has two phases. In phase 1, colors are assigned to vertices owned by each thread (i.e., to each partition \( P \in \mathcal{P} \)) separately without considering other partitions (\( \mathcal{P} \) denotes a set of all partitions). The maximum number of available colors can be specified as a parameter \( C \). In phase 2, border vertices (i.e., vertices with at least one edge leading to another partition; they form a set \( \mathcal{B} \) ) are verified for conflicts. If there are any, the colors are reassigned. This may cause conflicts within partitions, which are resolved during the next iteration. More iterations \( L \) may improve a solution (fewer colors used).

**Pushing and Pulling?** Both can be used in phase 2. For every border vertex \( v \), each \( u \in N(v) \) (\( t[u] \neq t[v] \) is analyzed. If \( u \) and \( v \) share the assigned color, then either \( u \)’s or \( v \)’s color is scheduled for a change (the update is pushed to or pulled from \( N(v) \)).

### 3.7 Minimum Spanning Tree (MST)

The goal of MST is to derive a spanning tree of \( G \) with the lowest sum of the included edge weights. The classical sequential algorithms: Prim [13] and Kruskal [13] lack parallelism. Therefore, we focus on the Boruvka [8] algorithm (more details on pushing and pulling in Prim and Kruskal are still provided in the technical report).

In Boruvka, each vertex is first associated with its own supervertex. In each iteration, two incident supervertices are merged into one along an edge \( e_m \) of a minimum weight. The algorithm proceeds until there is only one supervertex left. The selected minimum edges form the MST. MST algorithms are utilized in problems such as the design of broadcast trees [13].

**Pushing and Pulling in Boruvka?** First, selecting \( e_m \) adjacent to a given supervertex can be done by pushing (each supervertex overrides adjacent supervertices and their tentative minimal edges if it has a less expensive one) or by pulling (each supervertex picks its own \( e_m \)). Next, merging adjacent supervertices can also be done with pushing or pulling. Assume that each thread owns a number of supervertices. Now, it can either push the changes to the supervertices owned by other threads, or pull the information on the adjacent supervertices and only modify its own ones.

### 3.8 Push-Pull Insights

First, we present a generic difference between pushing and pulling. Recall that \( t[v] \) indicates the thread that owns \( v \). Define \( t \leadsto v \) to be true if \( t \) modifies \( v \) during the execution of a given algorithm (\( t \leadsto v \iff t[v] \)). Then

\[
(\text{Algorithm uses pushing}) \equiv \{ \exists r \in [1..T], v \in V \ t \leadsto v \land t \neq t[v] \}
\]

\[
(\text{Algorithm uses pulling}) \equiv \{ \forall r \in [1..T], v \in V \ t \leadsto v \Rightarrow t = t[v] \}
\]

In pushing, any thread \( t \) may access and modify any vertex \( v \in V \) so that we may have \( t \neq t[v] \). In pulling, \( t \) can only modify its assigned vertices: \( t[v] = t \) for any \( v \) modified by \( t \). In § 4, we show that this property determines that pulling requires less synchronization compared to pushing. However, pushing can often be done with less work, when only a subset of vertices needs to update its neighbors.

Second, our analysis shows that the push-pull dichotomy can be used in two algorithm classes: iterative schemes (PR, TC, GC, Boruvka MST) that derive some vertex properties and perhaps proceed in iterations until some convergence condition is met, and traversals (BFS, SSSP-\(\Delta \), BC).

### 4 THEORETICAL ANALYSIS

We now derive detailed specifications of push and pull algorithm variants and use them to investigate the differences between pushing and pulling. We (1) identify read and write conflicts, (2) conduct complexity analyses, and (3) investigate the amount of required atomics or locks. We focus on the CRCW-CB and CREW models. There exist past works on the parallel complexity of the considered algorithms [2, 6, 21, 30, 34, 37, 41]. Yet, we are the first to investigate the differences between pushing and pulling variants.

**Algorithm Listings** Our schemes have multiple variants as many nested loops can be parallel; we indicate them with \([n \text{ par}]\). Unless specified otherwise, we only consider the loops without square brackets in complexity analyses. We mark the read/write conflicts in the parts of the code related to pushing or pulling with \( R \) and \( W \), respectively. We indicate the data type in the modified memory cell to be either integer (\( I \)) or float (\( F \)). Finally, we use \( \text{grey backgrounds} \) to indicate pushing/pulling variants.

**Cost Derivations** We consider up to one processor per vertex, \( P \leq n \) (and \( P > d \)). Thus, pulling avoids write-conflicts, as each
thread accumulates updates for a given vertex. Still, pushing can update the same vertices multiple times at every iteration.

We formulate cost analyses of all algorithms via the primitives $k$-RELAXATION and $k$-FILTER. $k$-RELAXATION corresponds to simultaneously propagating updates from/to $k$ vertices to/from one of their neighbors for pushing/pulling. $k$-FILTER is used to extract the vertices updated in one or more $k$-RELAXATIONS, and is non-trivial only when pushing updates. We let $k = \max(1, k/P)$ and quantify the cost of these primitives. When pushing, $k$-RELAXATION takes $O(\hat{k})$ time and $O(\hat{k})$ work. A $k$-FILTER invocation requires $O(\log(\hat{P}) + \hat{k})$ time and $O(\min(k, n))$ work via a prefix sum.

When pushing, the cost of $k$-RELAXATION depends on the PRAM model. In the CRCW-CB model, $k$-RELAXATION takes $O(\hat{k})$ time and $O(\hat{k})$ work. In the CREW model, $k$-RELAXATION can be processed in $O(\hat{k}\log(\hat{d}))$ time via binary-tree reductions. To update each vertex of degree $d$ in the CREW model, we use a binary merge-tree with $d$ leaves. Over all trees, at most $k$ of $m$ leaves contain actual updates. We can avoid work for all nodes that are the roots of subtrees that do not contain updates, effectively computing a forest of incomplete binary trees with a total of $k$ leaves and maximum height $O(\log(d))$.

Each of $P$ processors propagates $k/P$ updates up the complete binary merge-tree associated with its vertices (requiring no setup time) in $O(\hat{k}\log(\hat{d}))$ time with a total of $O(\hat{k}\log(\hat{d}))$ work.

### 4.1 PageRank

PR (Algorithm 1) performs $O(L)$ steps of power iteration. For each step of power iteration, $k$-RELAXATION is called for $\hat{k} \in \{1, \ldots, \hat{d}\}$ with $\hat{d} = \sum_{i=1}^{\hat{d}} = m$. Thus the PRAM complexities of PR are $(1) O(L(m/P + \hat{d}))$ time and $O(\hat{Lm})$ work using pulling, $(2) O(L(m/P + \hat{d}))$ time and $O(\hat{Lm})$ work in pushing in CRCW-CB, and $(3) O(\log(\hat{d})(m/P + \hat{d}))$ time and $O(\hat{Lm}\log(\hat{d}))$ work using pushing in CREW.

**Conflicts** Pushing/pulling entail $O(Lm)$ write/read conflicts.

**Atoms/Locks** Pulling does not require any such operations. Contrarily, pushing comes with write conflicts to floats. To the best of our knowledge, no CPUs offer atomics operating on such values. Thus, $O(\hat{Lm})$ locks are issued.

### 4.2 Triangle Counting

TC is shown in Algorithm 2; this is a simple parallelization of the well-known Nodellerath scheme [44]. It employs $k$-RELAXATION for $\hat{k} \in \{1, \ldots, \hat{d}\}$ with $\sum_{i=1}^{\hat{d}} = O(m\hat{d})$. Thus the PRAM complexities of TC are $(1) O(\hat{d}(m/P + \hat{d}))$ time and $O(\hat{md})$ work using pulling, $(2) O(\hat{d}(m/P + \hat{d}))$ time and $O(\hat{md})$ work using pushing in CRCW-CB, and $(3) O(\hat{d}\log(\hat{d})(m/P + \hat{d}))$ time and $O(\hat{md}\log(\hat{d}))$ work using pushing in CREW. One can leverage more than $n$ processors to lower the PRAM time-complexity of TC [47].

**Conflicts** Both variants generate $O(\hat{md})$ read conflicts; pushing also has $O(\hat{md})$ write conflicts.

**Atoms/Locks** We use FAA atomics to resolve write conflicts.

### 4.3 Breath-First Search

BFS is shown in Algorithm 3. We define a generalized version of BFS, where vertices enter the frontier only after a given number of neighbors have been in the frontier. The standard BFS is obtained by setting this number to 1, but to use BFS from within BC, we will employ a counter specific to each vertex. The BFS pseudo-code also employs a given accumulation operator to compute values for each vertex as a function of values of its predecessors in the BFS tree.

Our analysis assumes this operator is commutative and associative. The frontier $F$ is represented as a single array while $my_F$ is private for each process and contains vertices explored at each iteration. All $my_F$ are repeatedly merged into the next $F$ (Line 8). We let $f_l$ be the size of $F$ in the $l$th iteration of the whole loop.

The call to $\text{explore}\_my_F$ in pulling requires checking all edges, so it takes $O(m/P + \hat{d})$ time and $O(m)$ work. The call to $\text{explore}\_my_F$ in pushing needs $O(\hat{d})$ consecutive $f_l$-RELAXATIONS, so it takes $O(\hat{f}_l\hat{d})$ time where $\hat{f}_l = \max(1, f_l/P)$ and work $O(\hat{f}_l\hat{d})$ in CRCW-CB (and $O(\log(\hat{d}))$ more in CREW). Second, the merge of frontiers can be done via a $\hat{d}f_l$-RELAXATION and, in pushing, a $\hat{d}f_l$-FILTER. The $\hat{d}f_l$-FILTER is not required in pulling, since we check whether
each vertex is in the frontier anyway. In pushing, the merge requires $O(\log(P) + d_f/P)$ time and $O(\min(d_f, n))$ work.

Thus, for a graph of diameter $D$ (with $D$ while-loop iterations) we derive the total cost using the fact that
\[(1)\ O(D(m/P + \bar{d}))\ time\ and\ O(Dm)\ work\ in\ pulling,\ (2)\ O(m/P + D(\bar{d} + \log(P)))\ time\ and\ O(m)\ work\ in\ pushing\ in\ CRCW,\ and\ (3)\ a\ factor\ of\ O(\log(d))\ more\ time\ and\ work\ in\ the\ CREW\ model.\ It\ is\ possible\ to\ achieve\ a\ lower\ time-complexity\ for\ BFS,\ especially\ if\ willing\ to\ sacrifice\ work-efficiency\ [18].

**Conflicts**

There are $O(m)$ write conflicts in pushing; pulling involves $O(Dm)$ read conflicts.

**Atoms/locks**

In pushing, each edge relaxation can be performed via a CAS atomic (in total $O(m\Delta)$ of these).

### 4.5 Betweenness Centrality

BC is illustrated in Algorithm 5. For each source vertex, we first compute a BFS to count the multiplicities of each shortest path and store all predecessors that are on some shortest path for each destination vertex. The list of predecessors is then used to define a shortest path tree. To calculate the partial centrality scores, this tree is traversed via BFS starting from the tree leaves. We use the ready array to ensure tree-nodes enter the frontier only once the partial centrality updates of all of their children are accumulated.

This algorithm (parallel Brandes) was described in detail [9, 34]. The approach is dominated by $2n$ BFS invocations, the cost of which is analyzed in § 4.3. For directed graphs, SSPS (e.g., Λ-stepping) must be used to compute each shortest-path tree. Given the shortest-path tree the partial centrality scores can be computed via BFS in the same way as for undirected graphs. Computationally, the most significant difference of BC from SSPS and BFS, is the presence of additional parallelism. Many source vertices can be processed independently, so up to $O(n^2)$ processors can be used by running $n$ independent instances of BFS or SSPS.

**Conflicts and Atoms/locks**

The number of conflicts as well as atoms or locks matches that of BFS or SSPS and can vary by the factor of up to $O(n)$ (depending on the amount of additional parallelism). Yet, since the accumulation operator for the second BFS (parallelism). Yet, since the accumulation operator for the second BFS uses floating point numbers, locks are required instead of atoms. This can be alleviated by maintaining sets of successors instead of predecessors as proposed by Bader et al. [3], which we identify as another opportunity for using either pushing or pulling. We elaborate on it in the technical report.

### 4.6 Boman Graph Coloring

We present BGC in Algorithm 6. The algorithm proceeds for $L$ iterations, a quantity that is sensitive to both the schedule of threads and the graph structure. To limit the memory consumption, we bound the maximum count of colors to $C$. We use an opaque function that partitions $G$ and thus initializes the set of border vertices $B$ and all the partitions $\mathcal{P} = \{P_1, \ldots, P_p\}$. The algorithm alternates between doing sequential graph coloring (seq_color_partition) and adjusting colors of bordering vertices. The adjustment of colors of bordering vertices corresponds to an invocation of $[\mathcal{B}]$-relaxation, in the worst case $[\mathcal{B}] = \emptyset(n)$. Therefore, the complexity of BGC is $(1) O(L(m/P + \bar{d}))$ time and $O(Lm)$ work using
pulling, (2) \(O(L(mP + \log \hat{d}))\) time and \(O(Lm)\) work using pushing in CRCW-CB, (3) \(O(\log(d))\) more than (2) using pushing in CREW.

**Conflicts** Pushing/pulling require \(O(Lm)\) write/read conflicts.

**Atomics/Locks** In pushing and pulling the write conflicts can be resolved via CASes (a total of \(O(Lm)\) of them).

---

We discuss some further extensions to our cost analyses. Please note that due to space constraints, several additional analyses can be found in the technical report.

**Atomics/Locks** In pushing and pulling the write conflicts can be resolved via CASes (a total of \(O(Lm)\) of them).

---

**4.7 Boruvka Minimum Spanning Tree**

Push- and pull-based Boruvka is shown in Algorithm 7. Due to space constraints, it only displays pushing/pulling when selecting the minimum edge adjacent to each supervertex. The algorithm starts with \(n\) supervertices and reduces their number by two at every iteration. The supervertex connectivity graph can density throughout the process with supervertices having degree \(O(n)\). However, the supervertices will always contain no more than \(m\) edges overall. Determining the minimum-weight edge for all supervertices requires \(O(n^2/P)\) time and \(O(m)\) work assuming each supervertex is processed sequentially. Merging the vertices requires \(O(log(n))\) time and \(O(n)\) work via a tree contraction [19] (our implementation uses a more simplistic approach). Merging the edges connected to each vertex can be done via \(O(n)\) invocations of a \(\Delta\)-relaxation, where \(k = O(n)\) at the first iteration and then the bound decreases geometrically. Over all \(\log(n)\) steps, the complexity of Boruvka is in \(O(n^2/P)\) and \(O(n^2)\) work using pushing, \(O(n^2/P)\) time and \(O(n^2)\) work using pushing in CRCW-CB, (3) \(O(\log(n))\) more than (2) using pushing in CREW.

Theoretically, known PRAM algorithms for finding connectivity and minimal spanning forests [1] are much faster in time complexity. Still, our simple scheme is fairly efficient in practice as supervertex degree generally grows much slower than in the worst case.

**Conflicts** Pushing/pulling require \(O(n^2)\) write/read conflicts.

**Atomics/Locks** The write conflicts in pushing can be handled via CAS atoms (in total \(O(n^2)\) of them).

---

**4.8 Further Analytical Considerations**

We discuss some further extensions to our cost analyses. Please note that due to space constraints, several additional analyses can be found in the technical report.

**More Parallelism** Our analysis considered parallelism with \(P \leq O(n)\). However, our pseudocodes specify additional potential sources of parallelism in many of the algorithms. Up to \(m\) processors can be used in many cases (and even more for TC), but in this scenario, the distinction between pushing and pulling disappears.

**Directed Graphs** Pushing and pulling differ interestingly for directed graphs. Pushing entails iterating over all outgoing edges of a subset of the vertices, while pulling entails iterating over all incoming edges of all (or most) of the vertices. Thus, instead of \(\hat{d}\) some cost bounds would depend on \(d_{out}\) and \(d_{in}\) for pushing and pulling, respectively; more details are in the technical report.

**4.9 Discussion & Insights**

We finally summarize the most important insights.

**Write/Read Conflicts** Pushing entails more write conflicts that must be resolved with locks or atomics (read conflicts must be resolved only under the EREW model). An exception is BC where the difference lies in the type of the data that causes conflicts (floats for pushing and integers for pulling as was remarked in the past work [34]). Moreover, traversals (BFS, BC (Part 2), SSSP) entail more read conflicts with pulling (e.g., \(O(\Delta\Delta d)\) in the BFS based on pulling and none in the push-based BFS).

**Atomics/Locks** We now summarize how conflicts translate into used atomics or locks. In many algorithms, pushing removes atomics or locks completely (TC, PR, BFS, \(\Delta\)-Stepping, MST). In others (BC), it changes the type of conflicts from \(\{\text{atomic}\} \to \{\text{lock}\}\) enabling the utilization of atomics and removing the need for locks [26].

**Communication/Synchronization** The above analyses show that pulling reduces synchronization compared to pushing (e.g., fewer atomics in TC). In contrast, pushing limits communication (e.g., the number of memory reads in BFS).

**Complexity** Pushing in traversals (BFS, BC, SSP: \(\Delta\)) entails more time and work (e.g., see BFS). On the other hand, in schemes such as PR that update all vertices at every iteration, pulling avoid write conflicts. As a result, for PR and TC, pulling is faster than pushing in the PRAM CREW model by a logarithmic factor.

**5 ACCELERATING PUSHING & PULLING**

Our analysis in § 4 shows that most push- and pull-based algorithms entail excessive counts of atomics/locks and reads/writes, respectively. We now describe strategies to reduce both.

**Partition-Awareness (PA, in Pushing)** We first decrease the number of atomics by transforming the graph representation to limit memory conflicts. For this, we partition the adjacency array of each \(v\) into two parts: \(local\) and \(remote\). The former contains the neighbors \(w \in N(v)\) that are owned by \(T[v]\) and the latter groups the ones owned by other threads. All local and remote arrays form two
contiguous arrays; offsets for each array are stored separately. This increases the representation size from \(n + 2m\) to \(2n + 2m\) but also enables detecting if a given vertex \(v\) is owned by the executing thread (to be updated with a non-atomic) or if it is owned by a different thread (to be updated with an atomic). This strategy can be applied to PR, TC, and BGC. Consider PR as an example. Each iteration has two phases. First, each thread updates its own vertices with non-atomic. Second, threads use atomics to update vertices owned by other threads. Here, the exact number of atomics depends on the graph distribution and structure, and is bounded by \(0\) (if \(\forall v \in V \forall w \in N(v) [f[v] \neq f[w]]\) and \(2m\) (if \(\forall v \in V \forall w \in N(v) [f[v] = f[w]]\)). The former occurs if \(G = (V, E)\) is bipartite (i.e., \(V = U \cup W, U \cap W = \emptyset\)) and each thread only owns vertices from either \(U\) or \(W\). The latter occurs if each thread owns all vertices in some \(G\)’s connected component.

The number of non-atomics stays similar. We show this example in Algorithm 8. The overhead from a barrier (line 10) is outweighed by fewer write conflicts (none in line 8).

---

**Algorithm 8:** (§ 5) Using Partition-Awareness for push-based PageRank.

**Frontier-Exploit (FE, in Pushing/Pulling)** The number of excessive reads/writes can be reduced by accessing only a fraction of vertices in each iteration (the Frontier-Exploit strategy), similarly to BFS. For example, consider BGC. In each iteration, every vertex is verified for potential conflicts, entailing many memory reads, regardless of whether pushing or pulling is used. To reduce the number of such reads, a set of vertices \(F \subseteq V\) that form a stable set (i.e., are not neighbors) is selected at first and is marked with a specified color \(c_0\) (we denote different colors with \(c_i, i \in \mathbb{N}\)). Then, the algorithm enters the main loop. In each iteration \(i \geq 1\), all neighbors of vertices in \(F\) that have not yet been colored are assigned a color \(c_i\); at the end of each iteration, \(F\) is set to \(\emptyset\) and the newly marked neighbors become the elements of \(F\). While iterating, for each vertex \(v \in F\), if any of its neighbors \(u \in N(v)\) has the same color \(c_i\), then a conflict occurs and either \(v\) or \(u\) (depending on the selected strategy) is assigned a color \(c_{i+1}\) that was not used before. This scheme resembles a BFS traversal with multiple sources selected at the beginning and marked with a color \(c_0\), and a frontier constituted by vertices in \(F\). In pulling, the vertices in \(F\) look for their uncolored neighbors and mark them with \(c_i\). In pulling, uncolored vertices look for colored neighbors that are in \(F\).

**Generic-Switch (GS, in Pushing/Pulling)** Next, we use the idea of switching between pushing and pulling; we want to not only reduce communication, but also limit the iteration count. We refer to the strategy as Generic-Switch. As an example, consider the above-described BGC enhanced with Frontier-Exploit. Pushing itself results in the excessive number of iterations. This is because, when the number of vertices to be colored is low (our experiments indicate \(< 0.1n\)), threads often conflict with each other, requiring more iterations. Switching to pulling may prevent new iterations as no conflicts are generated. Yet, using pulling too early would entail excessive memory accesses (few vertices are colored).

---

1 //The code below corresponds to lines 19-10 in Algorithm 1.
2 for \(v \in V_G\) is a set of vertices owned by a thread different from the
3 //local one.
4 for \(v \in V_G\) = \(V_G \cap \emptyset = \emptyset\).
5 6 for \(v \in V_G\) do in par
7 for \(u \in N(v)\) do \{in par\}
8 new_pr \([u] = f_p(r\{v\}) \cap d(v)\)
9
10 barrier(); // A lightweight barrier to synchronize all threads.
11
12 for \(v \in V_G\) do in par
13 for \(u \in N(v)\) do \{in par\}
14 new_pr \([u] = f_p(r\{v\}) \cap d(v)\)

Algorithm 9: (§ 5) Example of Conflict-Removal with BGC.

6 PERFORMANCE ANALYSIS

Finally, we investigate the performance of push/pull variants and the described acceleration strategies. Due to a large amount of data we present and discuss in detail a small representative subset; the remainder is in the report (see the link on page 1).

**Selected Benchmarks & Parameters** We consider the push- and pull-based variants, strategies from § 5, strong- and weak-scaling, Hyper-Threading (HT), and static/dynamic OpenMP scheduling. Two types of synthetic graphs are used: power-law Kro-necker [31] and Erdős-Rényi [16] graphs with \(n \in \{2^{20}, \ldots, 2^{26}\}\) and \(d \in \{2^3, \ldots, 2^{10}\}\). We also use real-world graphs (Table 2) of various sparsities: low \(d\) and large \(D\) (road networks), low \(d\) and \(D\) (purchase graphs), and large \(D\) with low \(d\) (communities). The graphs have up to 268M vertices and 4.28B edges.

**Used Programming Models** We use threading to harness SM systems. For DM machines, we use Message Passing (MP, also denoted as Msg-Passing) and Remote Memory Access (RMA) [20]. In MP, processes communicate explicitly and synchronize implicitly with messages. In RMA, processes communicate and synchronize explicitly by accessing remote memories with puts, gets, or atoms, and ensuring consistency with flushes [20].

**Counted Events** We incorporate the total of nine performance counters for detailed analyses of: cache misses (L1, L2, L3), reads and writes, conditional/unconditional branches, and data/instruction TLB misses. We also manually count issued atomics and acquired locks. Memory operations and cache/TLB misses are important as many graph algorithms are memory-bound [5]. Branches were also shown to impact performance in graph processing [23]. Finally, in distributed settings we count sent/received messages, issued collective operations, and remote reads/writes/atomics.

**Experimental Setup and Architectures** We use the following systems to cover various types of machines:
### 6.1 Shared-Memory Analysis

We first analyze the differences in the SM setting. The representative PAPI data for selected schemes is in Table 1. For each scheme, we discuss in more detail the results for graphs with: large $\bar{d}$ and low $D$, and low $\bar{d}$ and large $D$.

**PageRank** PR results can be found in Table 3. In graphs with both high $\bar{d}$ (orc, ljn, poc) and low $\bar{d}$ (rca, am), pulling outperforms pushing by $\approx 3\%$ and $\approx 19\%$, respectively. The former requires no atomics, but its speedup is moderate as it also generates more cache misses and branches as it accesses various neighbors, requiring more random memory reads.

**Breadth-First Search** The results are similar to SSSP-$\bar{d}$; pushing outperforms pulling in most cases. This is most visible for rca (high $D$, low $\bar{d}$) due to many memory accesses.

**Minimum Spanning Trees** We illustrate the MST results in Figure 4. We analyze time to complete each of the three most time-consuming phases of each iteration: Find Minimum (FM; looking for minimum-weight edges), Build Merge Tree (BMT; preparing metadata for merging), and Merge (M; merging of subtrees). Now, pushing is faster than pulling in BMT and comparable in M. Yet, it is slower in the most computationally expensive FM. In summary,
performance trends are similar to those of TC: pushing is consistently slower (≈20 for \(T = 4\)) than pulling. This is because the latter entails no expensive write conflicts.

Figure 4: (§ 6.1) Illustration of the MST analysis, each subplot relates to a different phase (XC40, HT enabled, the orc graph, \(T = 16\)).

**Betweenness Centrality** The results for BC can be found in Figure 5. We present the running times of both BFS traversals and the total BC runtime. In each case, pushing is slower than pulling because of the higher amount of expensive write conflicts that entail more synchronization in both BC parts.

Figure 5: (§ 6.1) Illustration of the BC analysis (scalability, XC40, HT enabled, the orc graph, \(T = 16\)).

### 6.2 Acceleration Strategies

We now evaluate the acceleration strategies (§ 5).

**Partition-Awareness (PA)** We start with adding PA to PR (Table 6a). In graphs with higher \(d\) (orc, ljn, pok), pushing+PA outperforms pulling (by \(\approx24\%\)). This is because PA decreases atomics (by 7%) and comes with fewer cache misses (\(\approx30\%\) for L1, \(\approx34\%\) for L2, and \(\approx69\%\) for L3) than pulling. In sparser graphs (rca, am), surprisingly pushing+PA is the slowest (\(\approx205\%\) than pulling). This is because fewer atomics issued in pushing+PA (\(\approx4\%\)) are still dominated by more branches (\(\approx23\%\)), reads (\(\approx44\%\)), and cache misses (\(\approx53\%\) for L3). We conjecture that in graphs with high \(d\), PA enhances pushing as the latter entails more atomics that dominate the performance. This is visible as both variants reduce the number of cache misses if adjacency lists are long and use better cache prefencers. Then, for low \(d\), adjacency lists are short on average, giving more cache misses in pushing+PA and pushing, making pulling the fastest. The worst performance of pushing+PA is due to the synchronization overheads (it splits each iteration into two phases separated by a barrier) that are no longer compensated with more effective cache utilization.

**Frontier-Exploit (FE), Generic/Greedy-Switch (GS/GrS)** We now apply these strategies to BGC, ensuring the same number of colors for each coloring. All three strategies entail very similar (< 1% of difference) times to compute each iteration. Here, we select GrS and compare it to simple pushing/pulling; see Figure 1. Faster iterations are due to fewer memory accesses as predicted in § 5. Next, we show that the strategies differ in the number of iterations, see Table 6b. The largest iteration count (especially visible for orc/ljn) is due to FE. As predicted, this is because of conflicts. Both switching strategies reduce the iteration count.

<table>
<thead>
<tr>
<th>(G)</th>
<th>Push+PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>orc</td>
<td>557.985 425.928</td>
</tr>
<tr>
<td>pok</td>
<td>103.907 87.577</td>
</tr>
<tr>
<td>ljn</td>
<td>240.943 145.475</td>
</tr>
<tr>
<td>am</td>
<td>5.422 13.705</td>
</tr>
<tr>
<td>rca</td>
<td>5.422 13.705</td>
</tr>
</tbody>
</table>

Figure 6: (§ 6.2) Acceleration strategy analysis (SM, Daint, XC30, \(T = 16\)). Time per iteration (ms) for PageRank (the left table). Number of iterations to finish for BGC (the right table).

### 6.3 Distributed-Memory Analysis

We also conduct a distributed-memory analysis.

#### 6.3.1 PageRank

First, we use RMA for push- and pull-based PR. The former uses remote atomics (MPI\_Accumulate) to modify ranks. The latter read the ranks with remote gets (MPI\_Get). Next, we design PR with MP. Here, we use the collective MPI\_Alltoallv [38] to exchange the information on the rank updates among processes. This variant is unusual as it combines pushing and pulling: each process contributes to the collective by both providing a vector of rank updates (it pushes) and receiving updates (it pulls).

**Performance** The performance outcomes (strong scaling) can be found in Figure 3. MP consistently outperforms RMA (by \(>10\times\)); pushing is the slowest. This may sound surprising as MP comes with overheads due to buffer preparation. Contrarily to RMA, the communication updates must first be placed in designated send buffers. Yet, the used MPI\_Accumulate is implemented with costly underlying locking protocol. Next, pulling suffers from communication overheads as it fetches both the degree and the rank of each neighbor of each vertex.
6.3.2 Triangle Counting. Similarly to PR, we develop push- and pull-based TC with RMA and with MP. In pushing, we increase remote counters with an FAA. The MP-based TC uses messages to instruct which counters are augmented. To reduce communication costs, updates are buffered until a given size is reached.

**Performance** The results are in Figure 3. RMA variants always outperform MP; pushing is always faster than pushing (<1% for orc and ≤25% for ljm for P = 48). This is different from PR as the counters in TC are integer and the utilized RMA library offers fast path codes of remote atomic FAAs that access 64-bit integers. The MP variant is the slowest because of the communication and buffering overheads.

**Memory Consumption** Both RMA schemes fetch N(v) of each analyzed vertex v to check for potential triangles. This is done with multiple MPI_Geths, with two extremes: a single get that fetches all the neighbors, or one get per neighbor. The former requires the largest amount of additional memory (O(1) storage per process) but least communication overheads. The latter is the opposite.

6.4 Further Analyses

We now show that the relative differences between pushing and pulling do not change significantly when varying the used machine. We verify that PR comes with the most relevant difference; see Table 4. Results vary most in denser graphs (orc, pok, ljm); for example pushing outperforms pulling on Trivium while the opposite is true on Daint. Contrarily, the results are similar for rca and am. Thus, the overheads from branches, reads, and cache misses (that are the highest in graphs with lowest d̄) dominate performance.

<table>
<thead>
<tr>
<th>System</th>
<th>Push</th>
<th>Pull</th>
<th>Push+PA</th>
<th>Daint (XC40)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1426.96</td>
<td>191.34</td>
<td>373.134</td>
<td>6.199</td>
</tr>
<tr>
<td>Pull</td>
<td>1583.09</td>
<td>279.26</td>
<td>421.396</td>
<td>2.819</td>
</tr>
<tr>
<td>Push+PA</td>
<td>1285</td>
<td>190.54</td>
<td>400.634</td>
<td>8.549</td>
</tr>
<tr>
<td>Daint (XC40)</td>
<td>499.46</td>
<td>123.78</td>
<td>248.602</td>
<td>5.744</td>
</tr>
<tr>
<td>Pull</td>
<td>456.532</td>
<td>86.812</td>
<td>206.604</td>
<td>5.810</td>
</tr>
</tbody>
</table>

Table 4. (§ 6.4) Time to compute one iteration in PR [ms] (SM setting with full parallelism (HT enabled); Trivium, T = 8; Daint, XC40, T = 24).

6.5 Push-Pull Insights

We finally summarize the most important insights on the push-pull performance for the considered systems.

**Shared-Memory Settings** First, some algorithms are the fastest with pushing (SSSP-Δ, BFS, and PR for dense graphs) except for some data points (e.g., iteration 6 for orc in SSSP-Δ). This contradicts the intuition that pulling comes with less overheads from atoms. Yet, they either entail more reads that dominate performance (e.g., SSSP-Δ) or use cache prefetchers less effectively by not accessing contiguous structures (e.g., PR). The results for PR+PA illustrate that atoms do not always dominate performance; this can happen if effects such as cache misses become less dominant.

Second, SSSP-Δ on SM systems is surprisingly different from the variant for the DM machines presented in the literature, where pulling is faster [12]. This is because intra-node atomics are less costly than messages. Next, HT accelerates each considered scheme, maintaining the relative differences between pushing and pulling. Finally, several pulling schemes (in BGC and MST) are faster than their push counterparts.

**Distributed-Memory Settings** The choice of PR and TC illustrates that two algorithms with push and pull variants having similar algorithm designs may come with substantially different performance patterns. Intuitively, RMA should ensure highest performance in both PR and TC as both require the same MPI_Accumulate remote atomic function. Yet, the different operand type results in different underlying implementations and thus results. With the setting considered in this work, RMA and MP ensured best performance for TC and PR, respectively.

7 DISCUSSION

We now discuss various aspects of push and pull variants.

7.1 Push-Pull: Linear Algebra

Various graph algorithms can be expressed with linear algebra (LA) operations such as matrix-vector (MV) multiplication. It enables a concise specification by abstracting from details such as scheduling vertices for processing in the next iteration [27]. We now illustrate that it is possible to frame LA-based graph algorithms in push and pull variants.

**Brief Recap** A crucial notion is the adjacency matrix of G (denoted as A) that encodes G’s structure. The element in row i and column j of A equals 1 iff there is an edge from vertex j to vertex i, and equals 0 otherwise. For simplicity, we focus on unweighted graphs, but our conclusions apply to the weighted case.

The graph algorithms that we consider can be cast as matrix-vector multiplications (MVs) \( A \otimes x^{(k)} \), where \( x^{(k)} \) is the algorithm state in iteration \( k \) and \( \otimes \) is matrix-vector multiplication operator over an appropriate semiring. The adjacency matrix A is generally sparse, while \( x^{(k)} \) may or may not be sparse depending on the computation. For example, in PR, each \( x^{(k)} \) is dense, while in BFS, the sparsity of \( x^{(k)} \) depends on the number of vertices in the 4th frontier. We refer to the case when the vector is dense as SpMV, and when the vector is sparse, SpMSpV. The dichotomy between push and pull algorithm variants is mirrored by the dichotomy between the Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) representations of A.

A CSR representation stores each row of A contiguously. The \( i \)th row of A contains all vertices with an edge to vertex \( i \). Consequently, performing an SpMV in the CSR layout involves iterating over each row and multiplying each nonzero element in the row by appropriate entries of the vector. Thus, each entry of the output can be computed independently by a thread. This scheme is equivalent to pulling updates for each vertex. For SpMV, CSR (pulling) works extremely well, but for SpMSpV, it is not clear how to efficiently exploit the sparsity of the vector \( x^{(k)} \).

A CSC representation stores each column of A contiguously. The \( i \)th row of A contains all vertices with an edge from vertex \( i \). Consequently, performing an SpMV in the CSC layout involves iterating over each column and multiplying each nonzero element...
in the column by the same entry of the vector, while accumulating to different elements of the output vector. Here, atomics or a reduction tree are necessary to combine updates to each output vector element. This scheme is equivalent to pushing updates from each vertex, as each thread is naturally assigned a different column of \( A \) and nonzero entry of \( x^{(k)} \). For SpMSpV, CSC (pushing) facilitates exploiting the sparsity of the vector by simply ignoring columns of \( A \) that match up to zeros in \( x^{(k)} \).

### 7.2 Push-Pull: Programming Models

Push/pull differences depend on the programming model:

- **Threading/RMA** The difference lies in the used atomics. An example is TC: no atomics (pulling) and FAA (pushing).

- **MP (Point-to-Point Messages)** In iterative algorithms with fixed communication patterns (e.g., TC) pushing gives more speedup as pulling increases the message count. In traversals, pushing-pulling switching offers highest performance [4, 12].

- **MP (Collectives)** In collectives such as MPI_Alltoallv, all processes both push and pull the data, eliminating the distinction between these two.

### 7.3 Push-Pull: Code Complexity

Push and pull variants considered in this work come with similar code complexity. Still, pull schemes can be more challenging in achieving high performance. Consider the inner loop in PR where an analysis of push and pull in software engineering has also been conducted [56]. None of these works addresses graph processing.

### 7.4 Push-Pull: Gather-Apply-Scatter

Finally, we discuss the relationship between the push-pull dichotomy and the well-known Gather-Apply-Scatter (GAS) abstraction [22]. In GAS, one develops a graph algorithm by specifying the gather, apply, and scatter functions. They run in parallel for each vertex \( v \) and respectively: bring some data from \( v \)’s neighbors, use it to modify \( v \)’s value, and write the result to a data structure. We now describe two algorithms designed with GAS (SSSP and GC) [22] and show how to develop them with pushing or pulling.

- **SSSP** Here, each vertex \( v \) is processed in parallel by selecting \( v \)’s incident edge \( e \) that offers a path to the selected root \( s \) with the lowest distance. If \( s \) is lower than the current distance from \( v \) to \( s \), the value is updated accordingly and \( N(s) \) are scheduled for processing in the next iteration. Now, push or pull can be applied when \( v \) updates its distance to \( s \). In the former, a neighboring vertex that performed a relaxation in the previous iteration updates its neighbors (pushes the changes) with new distances. In the latter, each vertex scheduled for updates iterates over its neighbors (pulls the updates) to perform a relaxation by itself.

- **GC** Every vertex \( v \) collects the set of colors on \( N(v) \) to compute a new unique color. Next, the new colors are scattered among \( N(v) \). Any conflicting vertices are then scheduled for the color recomputation in the next iteration. This algorithm is a special case of BGC: each vertex constitutes a separate partition (i.e., \( V \cap U \in N(v) \) if \( v \neq t[u] \)). Thus, the same approach can be incorporated.

### 8 RELATED WORK

#### Push and Pull Algorithm Variants

Several graph algorithms that approach the pushing and pulling distinction have been proposed. The bottom-up (pull) BFS was described by Suzumura et al. [48] while Beamer et al. [4] introduced a direction-optimizing BFS that switches between top-down (push) and bottom-up (pull) variants. Madduri et al. [34] proposed several improvements to BC, one of which inverts the direction of modifications in the backward traversal to eliminate critical sections. Whang et al. [52] described pulling and pushing in PR. Finally, Chakaravarthy et al. [12] inverts the direction of message exchanges in the distributed \( \Delta \)-Stepping algorithm. All these schemes are solutions to single problems. We embrace and generalize them in the push-pull analysis.

#### Pushing/Pulling in Graph Frameworks

Various graph processing frameworks were introduced, for example PBGL [24], Pregel [35], GraphBLAS [36], Galois [29], HAMA [45], PowerGraph [22], GraphLab [32], and Spark [34]. Some use pushing and pulling in certain ways: by sending and receiving messages (Pregel), using the GAS abstraction (PowerGraph), switching between sparse and dense graph structures (Ligra [46]), switching the direction of updates in a distributed environment (Gemini [57]), using pushing and pulling in 3D task-partitioning [55], or pushing and pulling to/from disk [51]. Yet, none of them comes with an analysis on the push-pull dichotomy, focusing on the framework design. Finally, Doekemeijer et al. [15] list graph processing frameworks that have push- or pull-based communication. Our theoretical analysis and performance observations can serve to help better understand and improve graph processing frameworks.

#### Accelerating Strategies

The Grace framework [40] partitions the graph similarly to Partition-Awareness, but its goal is to reduce caching overheads instead of atomics in pushing. Ligar uses a scheme similar to Generic-Switch as it switches between sparse and dense graph representations [46]. Finally, Salihoglu et al. [42] enhance Pregel-based systems with various schemes. Among others, similarly to Greedy-Switch, they propose to switch from a Pregel-based distributed scheme to a sequential algorithm variant.

#### Pushing/Pulling outside Graph Processing

Borokhovich et al. [7] analyzed gossip algorithms in network coding for information spreading using push, pull, and exchange communication schemes. Swamy et al. [49] designed an asymptotically optimal push-pull method for multicasting over a random network. Intel TBB uses a push-pull protocol in its flow graphs, biasing communication to prevent polling and to reduce unnecessary retries [50]. An analysis of push and pull in software engineering has also been conducted [56]. None of these works addresses graph processing.

### 9 CONCLUSION

Graph processing has become an important part of various CS research and industry fields, including HPC, systems, networking, and architecture. Its challenges, described by Lumsdaine et al. almost 10 years ago [33], have still not been resolved and accelerating graph computations remains an important goal that must be attained for the ability to process the enormous amounts of data produced today.

In this work, we accelerate graph algorithms by deriving the most advantageous direction of graph updates out of the two options: pushing the updates from the private to the shared state, or pulling
the updates in the opposite direction. We illustrate in a detailed analysis that the Push-Pull (PP) dichotomy, namely using either pushing or pulling, can be applied to various algorithms such as triangle counting, minimum spanning tree computations, or graph coloring. We provide detailed specifications, complexity analyses, and performance data from hardware counters on which variant serves best each algorithm and why pushing and pulling differ. These insights can be used to improve various graph processing engines.

Furthermore, we identify that pushing usually suffers from excessive amounts of atomic/locks while pulling entails more memory reads/writes. We use generic strategies to limit the amount of both, accelerating the processing of road networks, citation graphs, social networks, and others.

Our analysis illustrates that the decision on using either pushing or pulling is not limited to merely applying updates in PageRank or sending messages in BFS, but is related to a wide class of algorithms, strategies, graph abstractions, and programming models. Our PP dichotomy can easily be generalized to other concepts related to graph processing, for example vectorization.

Acknowledgments. We thank Hussein Harake, Colin McMurtrie, and the whole CSCS team granting access to the Greina, Piz Dora, and Daint machines, and for their excellent technical support. Maciej Besta is supported by Google European Doctoral Fellowship.

REFERENCES