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

Maciej Besta[1], Michał Podstawski[2 4], Linus Groner[1], Edgar Solomonik[3], Torsten Hoefler[1]
[1] Department of Computer Science, ETH Zurich
[2] Perform Group Katowice, [4] Katowice Institute of Information Technologies
[3] 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-memory (SM) and distributed-memory (DM) systems.

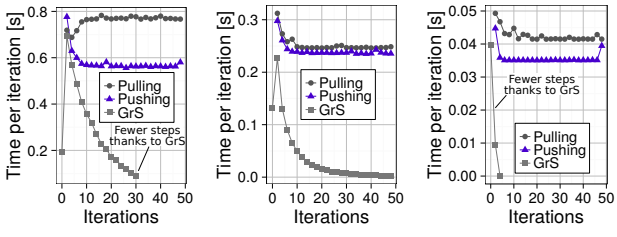
Direction optimization in breadth-first search (BFS) [4] is one of the mechanisms that have been used to alleviate these issues. It combines the traditional *top-down* BFS (where vertices in the active frontier iterate over all unvisited neighbors) with a *bottom-up* scheme (where unvisited vertices search for a neighboring vertex in the active frontier [49]). 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 both 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? Finally, when and how can pushing and 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 [47] and Gemini [57]. Yet, none of these works conducts an analysis on 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 by incorporating traversal-based graph coloring, and (2) switches between push- or pull-based scheme and an optimized greedy variant.



(a) Orkut network. (b) Livejournal graph. (c) CA road graph.

Figure 1: (§ 6.1) Boman graph coloring [6] results and (§ 6.2) the analysis of the strategy Greedy-Switch (GrS); single node of a Cray XC30, 16 threads.

We provide the following contributions:

- We apply the push-pull dichotomy to various classes of graph algorithms and obtain detailed formulations of multiple centrality schemes, traversals, calculating minimum spanning trees, graph coloring, and triangle counting. We also identify several existing graph processing schemes and

HPDC '17, June 26-30, 2017, Washington , DC, USA

ACM ISBN 978-1-4503-4699-3/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3078597.3078616

show that they are all 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 conduct performance analysis of push- and pull-based algorithms for both shared- and distributed-memory 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 various performance information using PAPI counters, such as the number of branches, cache misses, and issued atomic and non-atomic instructions.
- We incorporate strategies to reduce the amount of synchronization in pushing and memory accesses in pulling and illustrate that they accelerate various algorithms.
- We provide performance insights that can be used to enhance graph processing engines or libraries.
- Finally, we discuss whether the push-pull dichotomy is applicable in the algebraic formulation of graph algorithms.

2. MODELS, NOTATION, CONCEPTS

We first describe the necessary 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 *time* and *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 ($P' < P$) in time $S' = \lceil \frac{SP}{P'} \rceil$ for a fixed memory size M.

2.2 Graph Model, Layout, and Notation

We model an undirected graph G as a tuple (V, E); 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 $N(v)$ return the degree and the neighbors of a vertex v. The (non-negative) weight of an edge (v, w) is $\mathcal{W}_{(v,w)}$. We denote the maximum degrees for a given G as $\hat{d}$, $\hat{d}_{in}$ (in-degree), and $\hat{d}_{out}$ (out-degree). The average degree is denoted with a bar ($\overline{d}$). G's diameter is D.

To represent G, 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 rest of the system as if they occur instantaneously. They are used in lock-free graph computations to perform fine-grained updates [24,40]. In this work, we use CPU atomics that operate on integers. We now present the relevant operations:

Fetch-and-Add(*target, arg) (FAA): it increases *target by arg and also returns *target's previous value.

Compare-and-Swap(*target, compare, value, *result) (CAS): if *target == compare then *target = value and *result = true are set, otherwise *target is not changed and *result = false.

2.4 Communication & Synchronization

In the following, unless stated otherwise, we associate *communication* with: intra- or inter-node reads and writes, messages, and collective operations other than barriers. *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 algorithm that calculates the *rank* of each vertex v: $r(v) = (1-f)/|V| + \sum_{w \in N(v)} (f \cdot r(w)/d(w))$; f is the *damp factor* [10]. PR represents centrality schemes and is used to rank websites.

Pushing and Pulling? PR can be expressed in both push and pull variants [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 [44] 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 ($tc(v)$). 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 $tc(u)$ and $tc(w)$ while in the pull scheme it increments $tc(v)$.

3.3 Breadth-First Search (BFS)

The goal of BFS [13] is to visit each vertex in G. The algorithm starts with a specified *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 [40].

Pushing and Pulling? There exist both variants. The former is the traditional *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 (it pushes the updates from v to $N(v)$). The latter is the *bottom-up* approach [4, 49]: in each iteration every unvisited vertex u is tested if it has a parent in the frontier (the updates are pulled from $N(u)$ to u).

3.4 Single Source Shortest Path (SSSP)

In SSSP, one derives the distance from a selected source vertex s to all other vertices. We consider Δ-Stepping (SSSP-Δ) [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. Computing shortest paths has applications in, e.g., operations research.

Pushing and Pulling? Pushing and pulling is 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-Δ [12].

3.5 Betweenness Centrality (BC)

Intuitively, BC measures the importance of a vertex v based on the number of shortest paths that lead through v. Let σ_{st} be the number of shortest paths between two vertices s, t, and let $\sigma_{st}(v)$ be the number of such paths that lead through v. BC of v equals $bc(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Here, we consider Brandes' algorithm [9, 42]. Define the dependency of a source vertex s on v as: $\delta_s(v) = \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Then, we have $bc(v) = \sum_{s \neq v \in V} \delta_s(v)$ where $\delta_s(v)$ satisfies the following recurrence: $\delta_s(v) = \sum_{w:v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$; $pred(s, w)$ 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 $pred(s, v)$ and σ_{sv}, $\forall_{s,v \in V}$, obtaining a tree $\mathcal{T}$ over G. Next, $\mathcal{T}$ is traversed backwards (from the highest to the lowest distance) to compute $\delta_s(v)$ and $bc(v)$ based on the equations above. BC represents complex centrality schemes and is 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 frontier. 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 the vertices owned by each thread (i.e., to each partition $\mathcal{P} \in \mathscr{P}$) separately without considering other partitions ($\mathscr{P}$ denotes a set of all the partitions). The maximum number of available colors can be specified as a parameter $\mathcal{C}$. 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). GC has multiple applications in scheduling and pattern matching.

Pushing and Pulling? One can use pushing or pulling in phase 2. Here, for every border vertex v, each $u \in N(v)$ ($t[u] \neq t[v]$) is analyzed. If v and u have the same assigned colors, then either u's or v's color is scheduled for a change (the update is either 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 owned 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 $t \rightsquigarrow v$ to be true if t modifies v during the execution of a given algorithm ($t \rightsquigarrow v \Leftrightarrow t$ modifies v). Then

$$\text{(Algorithm uses pushing)} \Leftrightarrow \left(\exists_{t \in \{1..T\}, v \in V}\ t \rightsquigarrow v \wedge t \neq t[v]\right)$$

$$\text{(Algorithm uses pulling)} \Leftrightarrow \left(\forall_{t \in \{1..T\}, v \in V}\ t \rightsquigarrow v \Rightarrow t = t[v]\right)$$

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-Δ, 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 [2,6,21,30,34,37,42]. 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 `[in 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 Ⓡ/Ⓦ, respectively. We indicate the data type in the modified memory cell to be either integer (i) or float (f). Finally, we use grey backgrounds to indicate pushing/pulling variants.

Cost Derivations We consider up to one processor per vertex, $P \leq n$ (and $P > \hat{d}$). Thus, pulling avoids write-conflicts, as each thread accumulates updates for a given vertex. On the other hand, push-based schemes can generate multiple updates to the same vertices 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 $\bar{k} = \max(1, k/P)$ and quantify the cost of these primitives. When pulling, k-RELAXATION takes $\mathcal{O}(\bar{k})$ time and $\mathcal{O}(k)$ work. A k-FILTER invocation requires $O(\log(P)+\bar{k})$ 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 $\mathcal{O}(\bar{k})$ time and $\mathcal{O}(k)$ work. In the CREW model, k-RELAXATION can be processed in $\mathcal{O}(\bar{k}\log(\hat{d}))$ 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 $\mathcal{O}(\log(\hat{d}))$. Each of P processors propagates k/P updates up the complete binary merge-tree associated with its vertices (requiring no setup time) in $\mathcal{O}(\bar{k}\log(\hat{d}))$ time with a total of $\mathcal{O}(k\log(\hat{d}))$ work.

4.1 PageRank

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

Conflicts Pushing/pulling entail $\mathcal{O}(Lm)$ read/write conflicts, respectively.

Atomics/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, $\mathcal{O}(Lm)$ locks are issued.

4.2 Triangle Counting

TC is shown in Algorithm 2; this is a simple parallelization of the well-known NodeIterator scheme [45]. It employs k_i-RELAXATION for $i \in \{1,\ldots,\hat{d}^2\}$ with $\sum_{i=1}^{\hat{d}} = O(m\hat{d})$. Thus the PRAM complexities of TC are (1) $\mathcal{O}(\hat{d}(m/P+\hat{d}))$ time and $\mathcal{O}(m\hat{d})$ work using pulling, (2) $\mathcal{O}(\hat{d}(m/P+\hat{d}))$ time and $\mathcal{O}(m\hat{d})$ work using pushing in CRCW-CB, and (3) $\mathcal{O}(\hat{d}\log(\hat{d})(m/P+\hat{d}))$ time and $\mathcal{O}(m\hat{d}\log(\hat{d}))$ work using pushing in CREW. One can leverage more than n processors to lower the PRAM time-complexity of TC [48].

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

Atomics/Locks We can use FAA atomics to resolve all write conflicts.

```
1 /* Input: a graph G, a number of steps L, the damp parameter f
2   Output: An array of ranks pr[1..n] */
3
4 function PR(G,L,f) {
5   pr[1..v] = [f..f]; //Initialize PR values.
6   for(l = 1; l < L; ++l) {
7     new_pr[1..n] = [0..0];
8     for v ∈ V do in par {
9       update_pr(); new_pr[v] += (1 − f)/n; pr[v] = new_pr[v];
10 } } }
11
12 function update_pr() {
13   for u ∈ N(v) do [in par] {
14     {new_pr[u] += (f·pr[v])/d(v) Ⓦ f;}                    PUSHING
15
16     {new_pr[v] += (f·pr[u])/d(u) Ⓡ;}                      PULLING
17 } }
```

Algorithm 1: (§ 4.1) Push- and pull-based PageRank.

```
1 /* Input: a graph G. Output: An array of triangle counts
2  * tc[1..n] that each vertex belongs to. */
3
4 function TC(G) {tc[1..n] = [0..0]
5   for v ∈ V do in par
6     for w1 ∈ N(v) do [in par]
7       for w2 ∈ N(v) do [in par]
8         if adj(w1,w2) Ⓡ update_tc();
9   tc[1..n] = [tc[1]/2 .. tc[n]/2]; }
10 function update_tc() {
11   {++tc[w1]; /* or ++tc[w2]. */} Ⓦ i                     PUSHING
12
13   {++tc[v];}                                             PULLING
14 }
```

Algorithm 2: (§ 4.2) Push- and pull-based Triangle Counting.

4.3 Breadth-First Search

```
1 /* Input: a graph G, a set of ready counters and initial values
        R0 for each node, and an accumulation operator ⇐.
2  * Output: R[1..n] where R[F[i]]=R0[i] and other otherwise
        contains accumulation of all R values of predecessors. */
3
4 function BFS(G,ready,R0,⇐) {
5   my_F[1..P] = [∅..∅]; R = R0; F⊂ V, such that for each v ∈F,
        ready[v]=0;
6   while (F ≠ ∅)
7     explore_my_F(); {
8     F = my_F[1] ∪ my_F[2] ∪ .. ∪ my_F[P];  } }
9
10 function explore_my_F() {
11   for v ∈ F do in par                                    PUSHING
12     for w ∈ N(v) do [in par]
13       if ready[w] > 0 Ⓡ
14         R[w] ⇐ R[v] Ⓦ;
15     for w ∈ N(v) do [in par] {
16       ready[w]--;
17       if ready[w]==0 { my_F[p_ID] = my_F[p_ID] ∪ {w}; } }
18
19   for v ∈ V do in par {                                  PULLING
20     if ready[v] > 0 {
21       for w ∈ N(v) do [in par] {
22         if w ∈ F Ⓡ {
23           R[v] ⇐ R[w];
24           ready[v]--;
25           if ready[v] == 0 { my_F[p_ID] = my_F[p_ID] ∪ {v}; }
26 } } } }
```

Algorithm 3: (§ 4.3) Push- and pull-based Breadth-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`s are repeatedly merged into the next `F` (Line 8). We let f_i be the size of `F` in the ith iteration of the while loop.

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

Thus, for a graph of diameter D (with D while-loop iterations) we derive the total costs using the fact that $\sum_{i=1}^{D} f_i = n$, obtaining: (1) $\mathcal{O}(D(m/P + \hat{d}))$ time and $\mathcal{O}(Dm)$ work in pull-based schemes, (2) $\mathcal{O}(m/P + D(\hat{d} + \log(P)))$ time and $\mathcal{O}(m)$ work in push-based schemes in the CRCW model, and (3) a factor of $O(\log(\hat{d}))$ 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 $\mathcal{O}(m)$ write conflicts in pushing; pulling involves $\mathcal{O}(Dm)$ read conflicts.

Atomics/Locks Pushing requires $\mathcal{O}(m)$ CAS atomics.

4.4 Δ-Stepping SSSP

```
1 /* Input: a graph G, a vertex r, the Δ parameter.
2    Output: An array of distances d */
3
4 function Δ-Stepping(G, r, Δ){
5  bckt=[∞..∞]; d=[∞..∞]; active=[false..false];
6  bckt_set={0}; bckt[r]=0; d[r]=0; active[r]=true; itr=0;
7
8  for b ∈ bckt_set do { //For every bucket do...
9   do {bckt_empty = false; //Process b until it is empty.
10  process_buckets();} while(!bckt_empty); } }
11
12 function process_buckets() {
13  for v ∈ bckt_set[b] do in par                          PUSHING
14   if(bckt[v]==b && (itr == 0 or active[v])) {
15    active[v] = false; //Now, expand v's neighbors.
16    for w ∈ N(v) {weight = d[v] + W(v,w);
17     if(weight < d[w]) { R //Proceed to relax w.
18      new_b = weight/Δ; bckt[v] = new_b;
19      bckt_set[new_b] = bckt_set[new_b] ∪ {w};}
20     d[w] = weight; W i;
21     if(bckt[w]==b)R {active[w]=true; bckt_empty=true;}}} R
22  for v ∈ V do in par                                   PULLING
23   if(d[v] > b) {for w ∈ N(v) do {
24     if(bckt[w] == b && (active[w] or itr == 0)) {R
25      weight = d[w] + W(w,v) R;
26      if(weight < d[v]) {d[v]=weight; new_b=weight/Δ;
27       if(bckt[v] > new_b) {
28        bckt[v] = new_b; bckt_set = bckt_set ∪ {new_b};}
29       if(new_b == b) {active[v]=true; bckt_empty=true;}}}}}
30 }
```

Algorithm 4: (§ 4.4) Push- and pull-based Δ-Stepping SSSP.

The algorithm works in epochs. In each epoch, a bucket b is initialized with vertices whose tentative distances are $[(b-1)\Delta, b\Delta)$, and relaxations are computed until all vertices within distance $b\Delta$ are found. This means that in epoch b, edges are relaxed only from vertices whose final distances are within $[(b-1)\Delta, b\Delta)$.

Let L be the maximum weighted distance between any pair of vertices in the graph, and let l_Δ be the number of iterations done in any epoch. If n_i vertices fall into the ith bucket, at the ith epoch $\mathcal{O}(l_\Delta\hat{d})$ executions of n_i-RELAXATION will relax edges of vertices in the current bucket and up to l_Δ executions of n_i-FILTER will be used to update the set of vertices in the current bucket. So each edge will be relaxed $\mathcal{O}(l_\Delta)$ times. There are a total of L/Δ epochs, so the complexity of Δ-stepping is (1) $\mathcal{O}((L/\Delta)l_\Delta(m/P + \hat{d}))$ time and $\mathcal{O}((L/\Delta)ml_\Delta)$ work using pulling, (2) $\mathcal{O}(ml_\Delta/P + (L/\Delta)l_\Delta\hat{d})$ time and $\mathcal{O}(ml_\Delta)$ work using pushing in CRCW-CB, (3) $\mathcal{O}(\log(\hat{d}))$ more than (2) using pushing in CREW. Pushing achieves a smaller cost, since we relax the edges leaving each node in only one of L/Δ epochs. These results may be extrapolated to specific types of graphs considered in the original analysis [37].

Conflicts In pushing, there is a write conflict for each of $O(ml_\Delta)$ edge relaxations. In pulling, there is a read conflict for each of $O((L/\Delta)ml_\Delta)$ edge relaxations.

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

4.5 Betweenness Centrality

BC is illustrated in Algorithm 5. For each source vertex, the algorithm first computes a BFS to count the multiplicities of each shortest path and stores 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 (for pulling, lists of successors rather than predecessors should be stored). To calculate the partial centrality scores, this tree is traversed via BFS starting from the leaves of the tree. We make use of the `ready` array to ensure tree-nodes enter the frontier only once the partial centrality updates of all of their children have been accumulated.

This algorithm (parallel Brandes) was described in detail in various past sources [9, 34]. The approach is dominated by $2n$ invocations of BFS, the cost of which is analyzed in § 4.3. For directed graphs, it is necessary to use SSSP to compute each shortest-path tree, for which Δ-stepping can be used. 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 SSSP 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 SSSP.

Conflicts and Atomics/Locks The number of conflicts as well as atomics or locks matches that of BFS or SSSP and can vary by the factor of up to $O(n)$ (depending on the amount of additional parallelism utilized). However, since the accumulation operator for the second BFS uses floating point numbers, locks are required instead of atomics.

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 $\mathcal{C}$. We use an opaque function `init` that partitions G and thus initializes the set of border vertices $\mathcal{B}$ and all the partitions $\mathscr{P} = \{\mathcal{P}_1 ... \mathcal{P}_s\}$. 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

```
1 /* Input: a graph G. Output: centrality scores bc[1..n]. */
2
3 function BC(G) { bc[1..n] = [0..0]
4   Define Π so that any Π ∋ u = (index_u, pred_u, mult_u, part_u);
5   Define u ⇐_pred v with u, v ∈ Π so that u becomes
        u = (index_u, pred_u ∪ index_v, mult_u + mult_v, part_u);
6   Define u ⇐_part v with u, v ∈ Π so that u becomes
        u = (index_u, pred_u, mult_u, part_u + (mult_u/mult_v)(1 + part_v));
7
8   for s ∈ V do [in par] {
9     ready = [1, ..., 1]; ready[s] = 0;
10    R = BFS(G, ready, [(1, ∅, 0, 0)..(s, ∅, 1, 0)..(n, ∅, 0, 0)], ⇐_pred)];
11    Define graph G' = (V, E') where (u, v) ∈ E' iff index_v ∈ pred_u;
12    Let ready[u] be the in-degree of u ∈ V in G';
13    R = BFS(G', ready, R, ⇐_part);
14    for (index_u, pred_u, mult_u, part_u) ∈ R do [in par]
15      bc[u] += part_u; }
```

Algorithm 5: (§ 4.5) Push- and pull-based Betweenness Centrality.

of $|\mathcal{B}|$-RELAXATION, in the worst case $|\mathcal{B}| = \Theta(n)$. Therefore, the complexity of BGC is (1) $\mathcal{O}(L(m/P+\hat{d}))$ time and $\mathcal{O}(Lm)$ work using pulling, (2) $\mathcal{O}(L(m/P+\hat{d}))$ time and $\mathcal{O}(Lm)$ work using pushing in CRCW-CB, (3) $\mathcal{O}(\log(\hat{d}))$ more than (2) using pushing in CREW.

Conflicts Pushing/pulling require $\mathcal{O}(Lm)$ read/write conflicts, respectively.

Atomics/Locks In pushing and pulling the write conflicts can be resolved via CASes (a total of $\mathcal{O}(Lm)$ of these).

```
1 // Input: a graph G. Output: An array of vertex colors c[1..n].
2 // In the code, the details of functions seq_color_partition and
3 // init are omitted due to space constrains.
4
5 function Boman-GC(G) {
6   done = false; c[1..n] = [∅..∅]; //No vertex is colored yet
7   //avail[i][j]=1 means that color j can be used for vertex i.
8   avail[1..n][1..C] = [1..1][1..1]; init(B, P);
9   while (!done) {
10    for P ∈ P do in par {seq_color_partition(P);}
11    fix_conflicts(); } }
12
13 function fix_conflicts() {
14   for v ∈ B in par do {for u ∈ N(v) do
15     if (c[u] == c[v]) {
16       {avail[u][c[v]] = ∅ (W) i;}                    PUSHING
17
18       {avail[v][c[v]] = ∅ (R) i;}                    PULLING
19   }}
```

Algorithm 6: (§ 4.6) Push- and pull-based Boman Graph Coloring.

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 densify throughout the process with supervertices having degree $\Theta(n)$. 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 k-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 (1) $\mathcal{O}(n^2/P)$ time and $\mathcal{O}(n^2)$ work using pulling, (2) $\mathcal{O}(n^2/P)$ time and $\mathcal{O}(n^2)$ work using pushing in CRCW-CB, (3) $\mathcal{O}(\log(n))$ 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. However, the simple proposed algorithm is fairly efficient in practice as supervertex degree generally grows much slower than in the worst case.

Conflicts Pushing/pulling require $\mathcal{O}(n^2)$ write/read conflicts respectively.

Atomics/Locks The write conflicts in pushing can be handled via CAS atomics (in total $\mathcal{O}(n^2)$ of them).

```
1 function MST_Boruvka(G) {
2  sv_flag=[1..v]; sv=[{1}..{v}]; MST=[∅..∅];
3  avail_svs={1..n}; max_e_wgt=max_{v,w∈V}(W_(v,w) + 1);
4
5  while avail_svs.size() > 0 do {avail_svs_new = ∅;
6   for flag ∈ avail_svs do in par {min_e_wgt[flag] = max_e_wgt;}
7   for flag ∈ avail_svs do in par {
8    for v ∈ sv[flag] do {
9     for w ∈ N(v) do [in par] {
10      if (sv_flag[w] ≠ flag) ∧                        PUSHING
11         (W_(v,w) < min_e_wgt[sv_flag[w]]) (R) {
12        min_e_wgt[sv_flag[w]] = W_(v,w) (W) i;
13        min_e_v[sv_flag[w]] = w; min_e_w[sv_flag[w]] = v (W) i;
14        new_flag[sv_flag[w]] = flag (W) i; }
15      if (sv_flag[w] ≠ flag) ∧ (W_(v,w) < min_e_wgt[flag]) (R) {
16        min_e_wgt[flag] = W_(v,w); min_e_v[flag] = v;   PULLING
17        min_e_w[flag] = w; new_flag[flag] = sv_flag[w]; } (R)
18  } } }
19   while flag = merge_order.pop() do {
20     neigh_flag = sv_flag[min_e_w[flag]];
21     for v ∈ sv[flag] do sv_flag[flag] = sv_flag[neigh_flag];
22     sv[neigh_flag] = sv[flag] ∪ sv[neigh_flag];
23     MST[neigh_flag] = MST[flag] ∪ MST[neigh_flag]
24                     ∪ { (min_e_v[flag], min_e_w[flag]) }; } }
```

Algorithm 7: (§ 4.7) Push- and pull-based Boruvka MST.

4.8 Further Analytical Considerations

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

More Parallelism Our analysis considered parallelism with up to $\mathcal{O}(n)$ processors. 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 Directed graphs entail an interesting difference between pushing and pulling. 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 $\hat{d}_{out}$ and $\hat{d}_{in}$ 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., $\mathcal{O}(Dn\hat{d})$ in the BFS based on pulling and none in the push-based BFS).

Atomics/Locks We now summarize how the analyzed conflicts translate into used atomics or locks. In many algorithms, pulling removes atomics or locks completely (TC, PR, BFS, Δ-Stepping, MST). In others (BC), it changes the type of conflicts from f to i, 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 Pulling in traversals (BFS, BC, SSSP-Δ) entails more time and work, see BFS for an example. 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; we concretize each one with an example.

Partition-Awareness (PA, for 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 $u \in N(v)$ 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-atomics. 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 $\forall_{v \in V} \forall_{w \in N(v)} t[v] \neq t[w]$) and $2m$ (if $\forall_{v \in V} \forall_{w \in N(v)} t[v] = t[w]$). 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).

```
1 //The code below corresponds to lines 19-10 in Algorithm 1.
2 //V_L is a set of vertices owned by a local executing tread.
3 //V_G is a set of vertices owned by a tread different from the
4 //local one. V_L ∪ V_G = V; V_L ∩ V_G = ∅.
5
6 for v ∈ V_L do in par                          PART 1: LOCAL UPDATES
7   for u ∈ N(v) do [in par]
8     new_pr[u] += (f·pr[v])/d(v)
9
10 barrier(); //A lightweight barrier to synchronize all threads.
11
12 for v ∈ V_G do in par                          PART 2: REMOTE UPDATES
13   for u ∈ N(v) do [in par]
14     new_pr[u] += (f·pr[v])/d(v) (W) (I)
```

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

Frontier-Exploit (FE, for Pushing and Pulling) The number of excessive reads/writes can be reduced with imposing a BFS-like traversal over vertices so that only a fraction of vertices is accessed in each iteration (the *Frontier-Exploit* strategy). It can be used in BGC, PR, TC. As an example, consider BGC. In each iteration, every vertex is verified for potential conflicts, resulting in a significant number of 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 pushing, 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, for Pushing and Pulling) Next, we use the idea of switching between pushing and pulling in any graph algorithm; our goal is to not only reduce the number of memory accesses or messages sent, but also to *limit the number of iterations*. We refer to the strategy as *Generic-Switch*. As an example, consider the above-described BGC enhanced with Frontier-Exploit where in each iteration, a new frontier F of vertices are assigned colors, and iterations proceed until all the vertices are colored. Now, 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 as most vertices are not colored. Thus, one must carefully select a switching moment or strategy, for example switch if the ratio of the number of the colored vertices to the generated conflicts (in a given iteration) exceeds a certain threshold.

Greedy-Switch (GrS, for Pushing and Pulling) Generic-Switch may not always bring the desired speedups. For example, BGC with Frontier-Exploit may still need many iterations to color a small fraction of the remaining vertices due to many conflicts between threads that share vertices. In such cases, it is more advantageous to completely switch from a parallel variant (regardless of whether it does pushing or pulling) to an optimized greedy scheme.

Conflict-Removal (CR, for Pushing and Pulling) The final strategy (see Algorithm 9) completely removes conflicts in both pushing and pulling. Consider BGC as an example. Instead of solving conflicts over border vertices (the set $\mathcal{B}$) in each iteration, one can first use an optimized scheme (e.g., greedy sequential) to color them without any conflicts (thus, this scheme is advantageous if $|\mathcal{B}|$ is small compared to $|V|$). The remaining vertices can then be colored in parallel; no conflicts occur either as every $v \in \mathcal{B}$ is already colored.

```
1 //The code below corresponds to lines 9-11 in Algorithm 6.
2 seq_color_partition(B)
3 for P ∈ 𝒫 do in par {seq_color_partition(P);}
```

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

6. PERFORMANCE ANALYSIS

Finally, we investigate the performance of push/pull variants and show the advantages of 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 and Parameters We consider the push- and pull-based variants, strategies from § 5, strong-

Event	orc (PR)			rca (PR)			ljn (TC)		rca (TC)		orc (BGC)		rca (BGC)		pok (SSSP-Δ)		rca (SSSP-Δ)	
	Push	Push+PA	Pull	Push	Push+PA	Pull	Push	Pull	Push	Pull	Push	Pull	Push	Pull	Push	Pull	Push	Pull
L1 misses	335M	382M	572M	2,062M	10,560M	2,857M	10,815B	10,684B	4,290M	4,150M	3,599B	4,555B	76,117M	75,401M	54,57M	469M	11,01k	76,19M
L2 misses	234M	289M	446M	640k	7,037M	1,508M	700M	645M	2,303M	2,215M	3,656B	4,418B	74,48M	73,92M	50,74M	472M	9,46k	75,56M
L3 misses	64,75M	53,49M	181M	348M	537k	866k	439M	404M	1,075M	1,030M	36,94M	186M	229k	226k	8,52M	11,43M	308	279k
TLB misses (data)	130M	142M	129M	12,21k	274k	21628	66,44M	56,05M	37,45k	18,37k	229M	411M	4,046M	3,801M	3,763M	26,17M	403	513k
TLB misses (inst)	1188	336	1161	220	250	218	1090	660	214	233	141k	507k	510	577	1,984k	11,22k	71	370
atomics	234M	219M	0	5,533M	5,374M	0	1,066B	0	724k	0	0	0	0	0	0	0	0	0
locks	0	0	0	0	0	0	0	0	0	0	219M	219M	5,358M	5,358M	902k	44.60M	370	5.523M
reads	1,196B	1,183B	1,187B	43,39M	62,59M	37,49M	3,169T	3,158T	158M	135M	17,90B	23,04B	419M	404M	2.435B	2.339B	42,32k	454M
writes	474M	460M	237M	14.99M	14,86M	7,499M	10,71B	1,066B	18,97M	725k	3,866B	4,201B	97,44M	96,95M	718M	663M	9,545k	100M
branches (uncond)	234M	222M	1971	5,533M	7,340M	533	8,585B	616k	19,48M	631	2,714B	2,902B	67,58M	67,40M	441M	421M	5,171k	64.3M
branches (cond)	474M	466M	240M	15M	18,79M	9.467M	3,173T	3,173T	156M	156M	23,62B	32,46B	524M	495M	2.27B	2,192B	35,03k	518M

Table 1: (§ 6.1) PAPI events for PR, BGC (average per iteration), and TC, SSSP-Δ (total count) for the SM setting (Daint, XC30, $T = 16$).

and weak-scaling, static and dynamic OpenMP scheduling, and Hyper-Threading (HT). Two classes of synthetic graphs are used: power-law Kronecker [31] and Erdős-Rényi [16] graphs with $n \in \{2^{20}, ..., 2^{28}\}$ and $\overline{d} \in \{2^1, ..., 2^{10}\}$. We also use real-world graphs (Table 2) of various sparsities: low $\overline{d}$ and large D (road networks), low $\overline{d}$ and D (purchase graphs), and large $\overline{d}$ with low D (communities). The graphs have up to 268M vertices and 4.28B edges.

Type	ID	n	m	$\bar{d}$	$\bar{D}$
R-MAT graphs	rmat	33M-268M	66M-4.28B	2-16	19-33
Social networks	orc	3.07M	117M	39	9
	pok	1.63M	22.3M	18.75	11
Ground-truth [53] community	ljn	3.99M	34.6M	8.67	17
Purchase network	am	262k	900k	3.43	32
Road network	rca	1.96M	2.76M	1.4	849

Table 2: (§ 6) The analyzed graphs with skewed degree distributions.

Used Programming Models We use threading to harness the shared-memory (SM) parallelism. For distributed memories (DM), 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 atomics, 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:

- **CSCS Piz Daint (Daint)** is a Cray with various XC* nodes. Each XC50 compute node contains a 12-core HT-enabled Intel Xeon E5-2690 CPU with 64 GiB RAM. Each XC40 node contains an 18-core HT-enabled Intel Xeon E5-2695 CPU with 64 GiB RAM. We also used some XC30 nodes with an 8-core HT-enabled Intel E5-2670 Sandy Bridge CPU and 32 GiB RAM. The interconnection is based on Cray's Aries and it implements the Dragonfly topology [28]. The batch system is slurm 14.03.7. This machine represents massively parallel HPC machines.
- **Trivium V70.05 (Trivium)** is a server with Intel Core i7-4770 that has four 3.4 GHz Haswell 2-way multi-threaded cores. Each core has 32 KB of L1 and 256 KB of L2 cache. The CPU has 8 MB of shared L3 cache and 8 GB of RAM. This option represents commodity machines.

Infrastructure and Implementation Details We use the PAPI library (v5.4.1.1) to access performance counters. We spawn one MPI process per core (or per one HT resource if applicable). We use Cray-mpich (v.7.2.2) for MP and the foMPI library (v0.2.1) [20] for RMA. We also use OpenMP 4.0 and TBB from the Intel Programming Environment 6.0.3. We compile the code (with the -O3 flag) with g++ v4.9.2 (on Trivium) and Cray GNU 5.2.40 g++ (on CSCS systems).

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 $\overline{d}$ and low D, and low $\overline{d}$ and large D.

PageRank PR results can be found in Table 3. In graphs with both high $\overline{d}$ (orc, ljn, poc) and low $\overline{d}$ (rca, am), pulling outperforms pushing by ≈3% and ≈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.

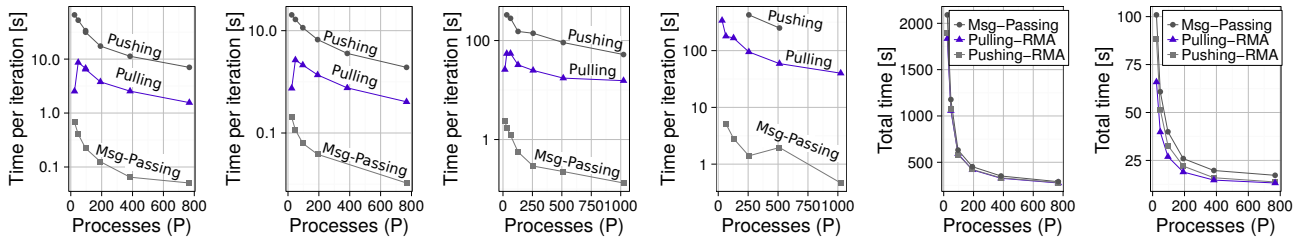
G	PageRank [ms]					Triangle Counting [s]				
	orc	pok	ljn	am	rca	orc	pok	ljn	am	rca
Pushing	572	129	264	4.62	6.68	11.78k	139.9	803.5	0.092	0.014
Pulling	557	103	240	2.46	5.42	11.37k	135.3	769.9	0.083	0.014

Table 3: (§ 6.1) Time per iteration for PageRank [ms] and the total time to compute for Triangle Counting [s] (SM setting, Daint, XC30, $T = 16$).

Triangle Counting We now proceed to TC (Table 3). Large amounts of time are due to the high computational complexity (§ 4.2); this is especially visible in graphs with high $\overline{d}$. Here, pulling always outperforms pushing (by ≈4% for orc and ≈2% for rca). This is due to atomics but also more cache misses that are caused by atomics.

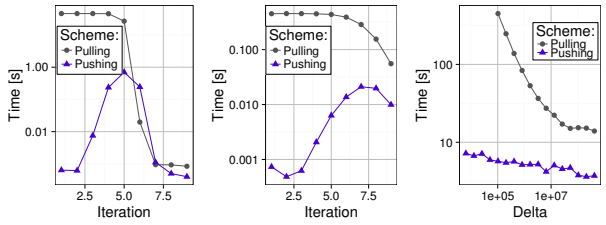
Graph Coloring The BGC results are presented in Figure 1. Pushing is always faster than pulling (by ≈10% for orc and ≈9% for rca for iteration 1). More detailed measurements indicate that the number of locks acquired is the same in both variants, but pushing always entails fewer cache/TLB misses and issued reads and writes.

Δ-Stepping The outcomes for orc and am can be found in Figure 2). Both push and pull variants use locks. Yet, a higher number of memory accesses issued in most iterations in the pull-based scheme limits performance. As expected, the difference decreases after several iterations because the frontier grows (with pushing), requiring more memory accesses. This is especially visible in graphs with high $\overline{d}$ where pulling outperforms pushing (e.g., iteration 6 for orc). Moreover, illustrate in Figure 2c that the larger Δ is, the smaller the difference between pushing and pulling becomes.



(a) PR, orc. (b) PR, ljn. (c) PR, rmat, $n = 2^{25}$. (d) PR, rmat, $n = 2^{27}$. (e) TC, orc. (f) TC, ljn.

Figure 3: (§ 6.3) The results of the scalability analysis in the DM setting (strong scaling, Daint, XC40, HT enabled, $T = 24$).

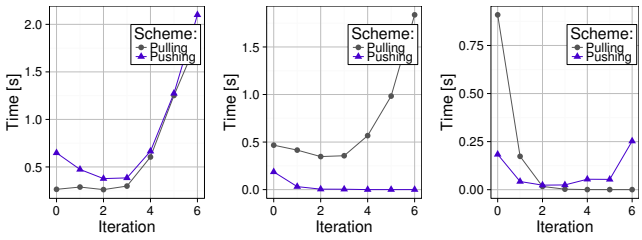


(a) The orc graph. (b) The am graph. (c) Varying Δ (orc).

Figure 2: (§ 6.1) SSSP-Δ SM analysis (XC30, HT enabled, $T = 16$).

Breadth-First Search The results are similar to SSSP-Δ; pushing outperforms pulling in most cases. This is most visible for rca (high D, low $\overline{d}$) 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 ($\approx$20 for $T = 4$) than pulling. This is because the latter entails no expensive write conflicts.



(a) "Find Minimum". (b) "Build Merge Tree". (c) "Merge".

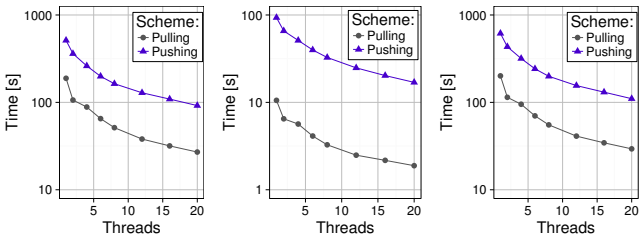
Figure 4: (§ 6.1) Illustration of the MST analysis, each subplot relates to a different phase (XC40, HT enabled, the orc graph, $T = 16$).

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.

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 $\overline{d}$ (orc, ljn, poc), pushing+PA outperforms pulling (by $\approx$24%). This is because PA decreases atomics (by 7%) and comes with fewer cache misses ($\approx$30% for L1, $\approx$34% for L2, and $\approx$69% for L3) than pulling. In sparser graphs (rca, am), surprisingly



(a) First BFS runtime (b) Second BFS runtime (c) Total runtime.

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

pushing+PA is the slowest ($\approx$205% than pushing). This is because fewer atomics issued in pushing+PA ($\approx$4%) are still dominated by more branches ($\approx$23%), reads ($\approx$44%), and cache misses ($\approx$53% for L3). We conjecture that in graphs with high $\overline{d}$, 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 prefetchers. Then, for low $\overline{d}$, 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 foe orc/ljn) is due to FE. As predicted, this is because of conflicts. Both switching strategies reduce the iteration count.

G	Push	+PA
orc	557.985	425.928
pok	103.907	87.577
ljn	240.943	145.475
am	2.467	5.193
rca	5.422	13.705

(a) Time per iteration (ms) for PageRank.

G	Push	+FE	+GS	+GrS
orc	49	173	49	49
pok	49	48	49	47
ljn	49	334	49	49
am	49	10	10	9
rca	49	5	5	5

(b) Number of iterations to finish for BGC.

Figure 6: (§ 6.2) Acceleration strategy analysis (SM, Daint, XC30, $T = 16$).

6.3 Distributed-Memory Analysis

We also conduct a distributed-memory analysis.

6.3.1 PageRank

First, we develop push- and pull-based PR with RMA. The former uses remote atomics (`MPI_Accumulate`) to modify ranks. The latter read the ranks with remote gets (`MPI_Get`). Second, we design PR based on MP. Here, we use the collective `MPI_Alltoallv` [39] to exchange the information on the rank updates among processes. This variant is unusual in that 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 >10x); pushing is the slowest. This may sound surprising as MP comes with overheads due to buffer preparation. Contrarily to RMA, the communicated 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.

Memory Consumption RMA variants only use $\mathcal{O}(1)$ storage (per process) in addition to the adjacency list. Contrarily, PR with MP may require up to $\mathcal{O}((n\hat{d})/P)$ storage (per process) for send and receive buffers.

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; pulling is always faster than pushing (<1% for orc and ≈25% for ljn 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_Gets`, 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 ($\mathcal{O}(\hat{d})$ 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, ljn); for example pushing outperforms pulling on Trivium while the opposite is true on Dora. 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 $\overline{d}$) dominate performance.

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 atomics. 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 atomics 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.

Trivium:	orc	pok	ljn	am	rca
Push	1426.966	191.340	373.134	6.199	16.818
Pull	1583.094	279.261	421.396	2.819	12.504
Push+PA	1289.123	190.541	400.634	8.549	52.068
Daint (XC40):					
Push	499.463	123.784	248.602	5.744	7.753
Pull	456.532	86.812	206.604	2.828	5.810
Push+PA	378.548	78.883	128.255	6.157	14.102

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

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 $\mathbf{A}$) that encodes G's structure. The element in row i and column j of $\mathbf{A}$ 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) $\mathbf{A} \otimes \mathbf{x}^{(k)}$, where $\mathbf{x}^{(k)}$ is the algorithm state in iteration k and $\otimes$ is matrix-vector multiplication operator over an appropriate semiring. The adjacency matrix $\mathbf{A}$ is generally sparse, while $\mathbf{x}^{(k)}$ may or may not be sparse depending on the computation. For example, in PR, each $\mathbf{x}^{(k)}$ is dense, while in BFS, the sparsity of $\mathbf{x}^{(k)}$ depends on the number of vertices in the kth 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 $\mathbf{A}$.

A CSR representation stores each row of $\mathbf{A}$ contiguously. The ith row of $\mathbf{A}$ 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 $\mathbf{x}^{(k)}$.

A CSC representation stores each column of $\mathbf{A}$ contiguously. The ith row of $\mathbf{A}$ 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 $\mathbf{A}$ and nonzero entry of $\mathbf{x}^{(k)}$. For SpMSpV, CSC (pushing) facilitates exploiting the sparsity of the vector by simply ignoring columns of $\mathbf{A}$ that match up to zeros in $\mathbf{x}^{(k)}$.

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 entails higher speedups as pulling increases the message count. In traversals, switching between pushing and pulling 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 a thread iterates over $N(v)$ of a given v. In pushing, updates are conducted simply with atomics. Contrarily, in pulling, one must also fetch the degrees of neighbors. This is similar for other pull variants and poses more challenges in making the code fast.

7.4 Push-Pull: Gather-Apply-Scatter

Finally, we discuss the relationship between the push-pull dichotomy and the well-know 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 it 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., $\forall_{v \in V} \forall_{u \in N(v)} t[v] \neq t[u]$). 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. [49] 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 Δ-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 [46], PowerGraph [22], GraphLab [32], and Spark [54]. 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 [47]), 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 [41] partitions the graph similarly to the Partition-Awareness scheme, but its goal is to reduce caching overheads instead of atomics in pushing. Ligra uses a scheme similar to Generic-Switch as it switches between sparse and dense graph representations [47]. Finally, Salihoglu et al. [43] 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. [50] 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 [38]. 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 atomics/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.

10. REFERENCES

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, 36(10):1258–1263, 1987.

[2] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Par. and Dist. Proc. Symp. (IPDPS)*, page 39. IEEE, 2004.

[3] D. A. Bader et al. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[4] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[5] S. Beamer, K. Asanović, and D. Patterson. GAIL: the graph algorithm iron law. In *Workshop on Ir. App.: Arch. and Alg.*, page 13, 2015.

[6] E. G. Boman et al. A scalable parallel graph coloring algorithm for distributed memory computers. In *Euro-Par*, pages 241–251. 2005.

[7] M. Borokhovich et al. Tight bounds for algebraic gossip on graphs. In *Inf. Theory Proc. (ISIT), IEEE Intl. Symp. on*, pages 1758–1762, 2010.

[8] O. Boruvka. O jistém problému minimálním. 1926.

[9] U. Brandes. A faster algorithm for betweenness centrality. *J. of Math. Sociology*, 25(2):163–177, 2001.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of Intl. Conf. on World Wide Web*, WWW7, pages 107–117, 1998.

[11] U. Catalyurek and C. Aykanat. A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices. In *Proc. of the Intl. Par. &Amp; Dist. Proc. Symp.*, IPDPS '01, pages 118–, 2001.

[12] V. T. Chakaravarthy et al. Scalable single source shortest path algorithms for massively parallel systems. In *Par. and Dist. Proc. Symp., IEEE Intl.*, pages 889–901, 2014.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[14] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

[15] N. Doekemeijer and A. L. Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, 2014.

[16] P. Erdős and A. Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi*, 2:482–525, 1976.

[17] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. of ACM Symp. on Theory of Comp.*, pages 114–118, 1978.

[18] H. Gazit et al. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Inf. Proc. Let.*, 28(2):61–65, 1988.

[19] H. Gazit et al. Optimal tree contraction in the EREW model. In *Concurrent Computations*, pages 139–156. Springer, 1988.

[20] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of the ACM/IEEE Supercomputing*, SC '13, pages 53:1–53:12, 2013.

[21] A. Goel and K. Munagala. Complexity measures for map-reduce, and comparison to parallel computing. *arXiv preprint arXiv:1211.6526*, 2012.

[22] J. E. Gonzalez et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.

[23] O. Green, M. Dukhan, and R. Vuduc. Branch-Avoiding Graph Algorithms. *arXiv preprint arXiv:1411.1460*, 2014.

[24] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Par. Obj.-Or. Scientific Comp. (POOSC)*, page 2, 2005.

[25] T. J. Harris. A survey of PRAM simulation techniques. *ACM Comp. Surv. (CSUR)*, 26(2):187–206, 1994.

[26] Intel, Inc. 64 and IA-32 Architectures Software Developer's Manual, 2015.

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

[28] J. Kim et al. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Ann. Intl. Symp. on Comp. Arch.*, ISCA '08, pages 77–88, 2008.

[29] M. Kulkarni et al. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, PLDI '07, pages 211–222, 2007.

[30] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. of ACM Symp. on Par. in Alg. and Arch.*, pages 303–314, 2010.

[31] J. Leskovec et al. Kronecker graphs: An approach to modeling networks. *J. of Machine Learning Research*, 11(Feb):985–1042, 2010.

[32] Y. Low et al. Graphlab: A new framework for parallel machine learning. *preprint arXiv:1006.4990*, 2010.

[33] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Par. Proc. Let.*, 17(1):5–20, 2007.

[34] K. Madduri et al. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Par. & Dist. Proc. (IPDPS), IEEE Intl. Symp. on*, pages 1–8, 2009.

[35] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *Proc. of the ACM SIGMOD Intl. Conf. on Manag. of Data*, SIGMOD '10, pages 135–146, 2010.

[36] T. Mattson et al. Standards for graph algorithm primitives. *arXiv preprint arXiv:1408.0393*, 2014.

[37] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[38] Michael Voss (Intel). Understanding the Internals of tbb::graph : Balancing Push and Pull.

[39] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3, 2012.

[40] R. C. Murphy et al. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[41] V. Prabhakaran et al. Managing large graphs on multi-cores with graph awareness. In *USENIX Annual Technical Conference*, volume 12, 2012.

[42] D. Prountzos and K. Pingali. Betweenness centrality: algorithms and implementations. In *ACM SIGPLAN Notices*, volume 48, pages 35–46. ACM, 2013.

[43] S. Salihoglu and J. Widom. Optimizing graph algorithms on Pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[44] N. Satish et al. Navigating the maze of graph analytics frameworks using massive graph datasets. In *ACM SIGMOD Intl. Conf. on Man. of Data*, pages 979–990, 2014.

[45] T. Schank. *Algorithmic aspects of triangle-based network analysis*. PhD thesis, University Karlsruhe, 2007.

[46] S. Seo et al. HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *Intl. Conf. on Cloud Comp. Tech. and Science*, CLOUDCOM'10, pages 721–726, 2010.

[47] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146, 2013.

[48] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st Intl. Conf. on Data Engineering*, pages 149–160, April 2015.

[49] T. Suzumura et al. Performance characteristics of Graph500 on large-scale distributed environment. In *Workload Char. (IISWC), IEEE Intl. Symp. on*, pages 149–158, 2011.

[50] V. N. Swamy et al. An Asymptotically Optimal Push–Pull Method for Multicasting Over a Random Network. *Inf. Theory, IEEE Tran. on*, 59(8):5075–5087, 2013.

[51] Z. Wang et al. Hybrid Pulling/Pushing for I/O-Efficient Distributed and Iterative Graph Computing. In *ACM Intl. Conf. on Man. of Data*, pages 479–494, 2016.

[52] J. J. Whang et al. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. In *Euro-Par: Par. Proc.*, pages 438–450. 2015.

[53] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[54] M. Zaharia et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the USENIX Conf. on Net. Sys. Design and Impl.*, NSDI'12, pages 2–2, 2012.

[55] M. Zhang et al. Exploring the hidden dimension in graph processing. In *USENIX Symp. on Op. Sys. Des. and Impl. (OSDI 16)*, 2016.

[56] Y. Zhao. *A model of computation with push and pull processing*. PhD thesis, Citeseer, 2003.

[57] X. Zhu et al. Gemini: A computation-centric distributed graph processing system. In *USENIX Symp. on Op. Sys. Des. and Impl. (OSDI 16)*, 2016.