# Filtering Wasteful Vertex Visits in Breadth-First Search

Prachatos Mitra
Georgia Institute of Technology
Atlanta, GA, USA
pmitra32@gatech.edu

Alexandros Daglis
Georgia Institute of Technology
Atlanta, GA, USA
alexandros.daglis@cc.gatech.edu

## ABSTRACT

Breadth-First Search (BFS) is a common building block for several graph processing algorithms today. In this work, we highlight that a large fraction of vertex visits across the network in distributed BFS results in wasteful work. We investigate methods to identify and filter such wasteful cross-network vertex visits to save network bandwidth for energy and performance improvements. We analyze the metadata requirements to perform such filtering in modern hierarchical distributed architectures and identify the tradeoffs between storage and filtering rate. We perform our experiments using the graph500 benchmark and provide a model to scale results to larger graphs. Finally, we propose heuristics to reduce the storage for a BFS message filter and explore the design space for implementing such filtering logic in software, hardware, or a combination.

**ACM Reference Format:**
Prachatos Mitra and Alexandros Daglis. 2023. Filtering Wasteful Vertex Visits in Breadth-First Search. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3624062.3625133

## 1 INTRODUCTION

Graph processing algorithms are fundamental in several areas of computing, such as data mining, AI and social networks. Key computational kernels such as Breadth-First Search (BFS) form an important building block for graph processing algorithms and are included in several benchmark suites [5]. Implementing an efficient BFS algorithm for distributed highly parallel execution is challenging due to the nature of graph datasets. First, a large number of real-life graphs exhibit power law connectivity, resulting in irregular access patterns with low data reuse. More fundamentally, however, BFS algorithms tend to be dominated by data access over computation in the distributed setting. This leads to communication (i.e., data movement over the network) being the primary performance determinant [4].

Prior works optimize BFS by improving locality through techniques such as 2D decomposition [4, 6] or algorithm modifications to reduce computations [3]. However, all these approaches still result in more computation and communication than necessary, incurring redundant edge traversals that result in repeated visits to the same vertices, compared to an optimal approach that would only traverse an edge if the destination vertex has not yet been visited. Redundant vertex visits entail excess work that consumes additional compute cycles and—in the case of distributed settings—precious network bandwidth, which is often the main performance bottleneck. Importantly, the larger the graph, the higher the fraction of wasteful vertex visits. Therefore, early detection and trimming of such wasteful edge traversals can reduce overall compute and network bandwidth utilization for performance and energy gains.

We focus on methods to identify and filter unnecessary vertex visits to reduce the communication volume in BFS, investigating the challenges and opportunities arising from the hierarchical nature of many modern architectures. We highlight how hierarchical architectures can be leveraged to cull excess communications by aggregating information at intermediate levels and using it as an early filter that eliminates wasteful vertex visits by the source of the visit's initiation. We make the following contributions:

- We introduce a method of filtering wasteful vertex visits for BFS in hierarchical architectures and analyze the relation of required storage overheads versus achieved filtering rates.
- We propose a method of extrapolating results for filtering performance and storage overheads to massive graphs of sizes that are intractable for most current systems.
- We investigate a number of alternative software and hardware designs to identify practical filtering implementations that balance storage requirements versus achieved filtering rates.

**Scope and limitations:** While filtering can be applied to several graph algorithms, our analysis focuses on BFS. This short paper does not delve into specific hardware designs for filtering, but lays the groundwork to determine the relation between hardware requirements and filtering efficiency. The results of the study can help hardware designers determine the type of hardware and resource provisioning required as a function of the desired filtering effectiveness, and gauge the attainable energy/performance gains.

## 2 FILTERING OPPORTUNITY IN BFS

### 2.1 Hierarchical architectures

We focus on the execution of BFS on large distributed systems with a hierarchical structure consisting of thousands of interconnected hardware coherent islands, as shown in Fig. 1. We use the term **group** for a collection of processing elements (PEs) that define a shared memory domain and have direct access to a shared slice of memory. A "compute node" may comprise several groups sharing an endpoint that enables access to the distributed system's network (i.e., a NIC). Fig. 1 shows a compute node with two groups and two PEs per group. The groups are connected over a Network on Chip (NoC) to the system-level network through an aggregation point termed **group connector** (GC). Such an aggregation point features a network interface (NIC), which may feature compute capabilities.


*SC-W 2023, November 12–17, 2023, Denver, CO, USA*

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3625133

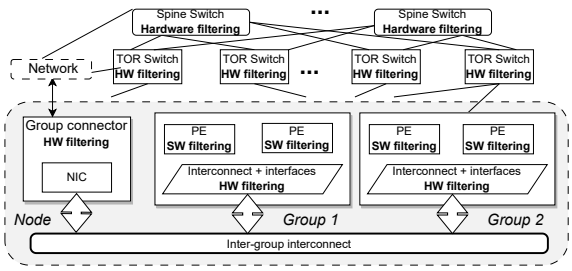



**Figure 1: High-level hierarchical architecture design with candidate locations for deployment of filtering mechanisms.**

Fig. 1 also marks candidate locations where filtering mechanisms can be deployed across the hierarchical architecture's multiple tiers. We can perform software filtering at each individual PE, or establish hardware filtering units at each group interface and GC. These hypothetical hardware filtering structures may be implemented in a network processor, leveraging their "gateway" position which naturally grants visibility of all messages entering and exiting a group, or a set of groups, depending on the tier. For simplicity, we focus on filtering opportunities at the different levels of a single compute node (groups and GCs), but all analyses can be extrapolated to the higher resource grouping levels introduced by the network's switches, as marked in Fig. 1.

We refer to a collection of compute resources at various levels as a **logical partition** (LP). For instance, a group defines an LP that encapsulates a number of PEs in the same shared memory domain, a GC defines an LP that encapsulates all PEs in the groups connecting to that GC, etc. The filtering rate achievable and the associated storage required is a function of the graph slice size contained within an LP. In a hierarchical system, the level where a filtering approach is applied defines the number of LPs the system is divided into. In Fig. 1's example, filtering at the lowest level of the hierarchy (i.e., individual PEs) results in the highest number of LPs (# nodes × 4). A filtering mechanism applied at the group level reduces the number of LPs by 2×, while doing so at the GC level reduces LPs by an additional 2×.

## 2.2 Vertex Visit Filtering in BFS

*2.2.1 Overview.* BFS is a building block for several graph algorithms. We consider a graph $G = (V, E)$ where $V$ is the set of vertices and $E$ is the set of bi-directional edges, and use the top-down BFS algorithm, where we start with a root vertex $v \in V$, and expand the frontier by adding each vertex $v_i$ that is connected to $v$. This is repeated in each step, with the frontier only comprising vertices not visited in a prior step. Parallel implementations of BFS are typically level-synchronous: each frontier may be processed in parallel in a superstep with a barrier between two frontiers. In the worst case, we perform checks on $|E|$ vertices, while the optimal number is $|V|$ since each vertex needs to be visited only once. This difference can significantly impact performance when $|E| >> |V|$.

We consider Kronecker-generated synthetic graphs from Graph500 [5] (of $2^{24}$ vertices and edge factor of 16, unless otherwise specified), with vertices distributed across the system's memory partitions uniformly at random. These graphs have a power law characteristic for their vertex degree and exhibit small-world and scale-free properties—similar to real-life graphs such as social networks—resulting in two effects. First, the frontier size grows rapidly, and is a significant fraction of the graph within a few supersteps. Second, the power law nature of the in-degrees leads to some vertices having a disproportionately high number of checks performed on them due to being connected to a large number of vertices. Therefore, in conventional top-down BFS on such graphs, a few such vertices cause a large number of wasteful checks.

*2.2.2 Filtering in BFS.* Fig. 2 illustrates the occurrence of wasteful vertex visits in BFS. Starting from vertex A, superstep 1 visits vertices B and C. Superstep 2 visits new vertices D and E, but D is visited twice, and vertex A is visited again. BFS concludes in superstep 3 with two visits to vertices A and F, of which the visit to A is wasteful. Ultimately, instead of the ideal number of 6 vertex visits required to complete BFS, 10 checks are performed when no form of filtering is present. A vertex visit in a single-node system incurs compute and memory access overheads, while in a distributed setting it may also incur data movement on the bandwidth-constrained network. Hence, filtering wasteful vertex visits can reduce computations and memory/network traffic.

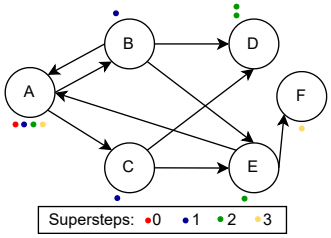



**Figure 2: Example of wasteful vertex visits in BFS.**

Table 1 shows the achievable filtering degree assuming an ideal mechanism that tracks *all prior vertex visits globally*: for a Kronecker graph with $2^{26}$ vertices and edge factor 16, almost 99% of vertex visits can be filtered. However, to reduce network traffic in a distributed processing setting, filtering must be done at the source of the vertex visit's initiation, which means each LP can only use locally available information, which is necessarily a small subset of the global graph information. As Fig. 3 shows, the maximum possible visits that can be filtered drops to 85% with 32 LPs for a graph with $2^{24}$ vertices. The amount of achieved filtering decreases almost linearly with the number of LPs—e.g., the opportunity of 85% filtering in a 32-LP system drops to a mere 13% on a 16K-LP system.

| Graph scale (Edge factor: 16) | Filtered vertex visits (%) |
|---|---|
| 20 | 97.1 |
| 22 | 97.5 |
| 24 | 98.2 |
| 26 | 98.8 |

**Table 1: Filtering rate vs. graph size.**

While the filtering opportunity drops with system size, it increases with graph size, as the power law nature of graph connectivity results in few highly connected vertices and many isolated vertices. Scaling the trend observed in Fig. 3 for graphs of size $2^{20} - 2^{27}$ linearly, we estimate ~ 60% filtering for a $2^{42}$ graph on a 16K-LP system, and 90% on a 2K-LP system. However, achieving such filtering rates assumes that we store information at each LP about *all* vertices of the graph that have been visited so far, which would introduce a storage requirement of 100s of GBs for a graph of that size, replicated at each LP. Thus, we study the limits of filtering possible as a function of storage requirements.

## 2.3 Filtering in hierarchical architecture

*2.3.1 Software filtering.* Filtering can be performed in software at each group or each PE by maintaining a list of all vertices that any message from the group or PE has visited in the past. As mentioned

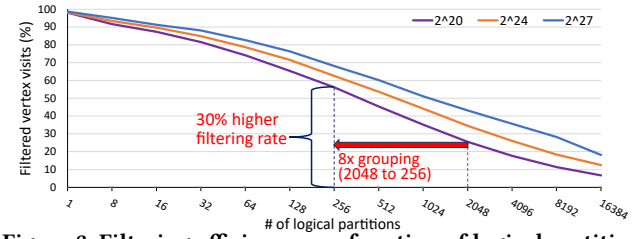



**Figure 3: Filtering efficiency as a function of logical partition count, for three different graph sizes.**

earlier, this can filter 60% of the total traffic on a system comprising 16K groups and operating on a graph with $2^{42}$ vertices.

*2.3.2 Hardware filtering.* Aggregating visited information across multiple groups and using it to perform filtering at a higher-level LP can increase the filtering rate. For example, the GC is a good candidate for performing filtering using aggregated information from all the groups located beneath it, thus forming a larger LP. Consider a system that applies filtering at the level of a GC that connects 8 groups together. Fig. 3 shows that, for a graph with $2^{20}$ vertices, reducing LPs by 8×, from 2048 to 256, via filtering information aggregation improves filtering rate by 30.5%.

## 3 STORAGE & FILTERING RATE TRADEOFFS

The primary design tradeoff for a BFS filtering logic is between the memory required to track previously visited vertices and the achievable filtering rate. A secondary aspect for hierarchical systems is the level in the hierarchy to apply filtering logic, which defines the resulting number of LPs. Finally, we need a method to scale analytical results that are obtained from smaller graph sizes to the largest graphs that may run on such systems.

### 3.1 Scaling filtering behavior to huge graphs

We construct scaling models to extrapolate results from small graphs we can run in our setup ($2^{27}$) to the largest size defined by graph500 ($2^{42}$).

*3.1.1 Storage scaling.* A straightforward approach to perform filtering is by maintaining a bit-vector for vertices previously visited at each LP [1]. Storage requirements (1 bit per vertex) scale linearly with the size of the graph as well as the number of LPs. For instance, a $2^M$ graph with N LPs needs a bit-vector with $2^M$ bits for each of the N LPs. Alternative approaches require the same percentage of the graph to be stored to achieve filtering across different graph sizes. For instance, selectively storing filtering information for vertices above a certain percentile category (e.g., based on vertex degree) will require storing a similar fraction of the graph regardless of the graph's size. Moreover, the fraction of vertices in each frontier of BFS does not significantly change with larger graph sizes [3]. Hence, the maximum frontier size as a fraction of the graph stays the same if we do not change the number of LPs while doubling the graph size. Therefore, we use a linear projection for filtering storage requirements with graph size as follows:

$$\text{Storage}(degree_{i+1}) = \text{Storage}(degree_i) \times 2^{(degree_{i+1} - degree_i)} \quad (1)$$

*3.1.2 Filtering ratio (FR) scaling.* The achievable FR increases with graph size: due to the graph's extreme power law, the number of isolated vertices goes up and few vertices with very high in-degree can get filtered (in-degree −1) times [2]. Fig. 3 shows slow linear scaling with an increase in graph size. For example, with 256 LPs, a $2^{27}$ graph has 65% filtering rate, while a $2^{24}$ graph has 60%, an 8% relative increase as the graph size grows by 8×. We consider this FR increase ratio to extrapolate the achieved FR to larger graphs (8% improvement with each increase in graph scale by 3) and define the scale factor $S$ ($S = \frac{8}{3}\%$), leading to the equation:

$$\text{FR}(degree_{i+1}) = \text{FR}(degree_i) \times (1 + (degree_{i+1} - degree_i) \times S) \quad (2)$$

Therefore, scaling from $2^{27}$ to $2^{42}$ increases the FR by 40% of the value at $2^{27}$, namely from 65% to an expected 91%.

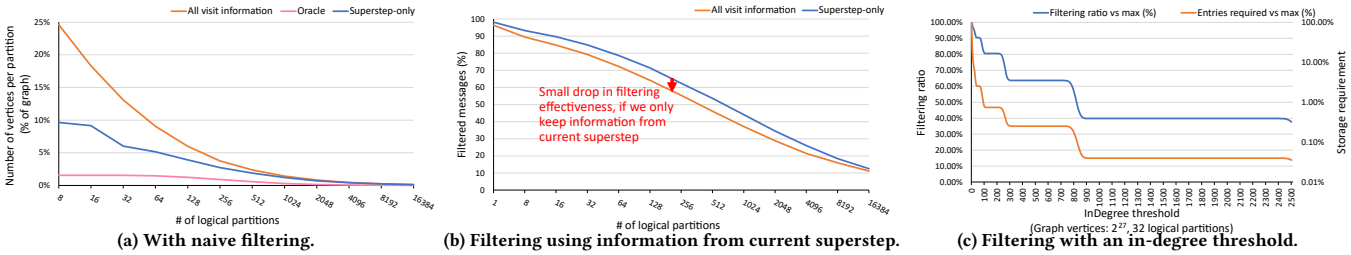
### 3.2 Baseline storage requirements

The simplest method of storing information about past visits is by using a bit-vector, thus requiring 512GB for a $2^{42}$ graph, which would be infeasible to store per LP, whether the LP is a PE or even a group. Our goal is to reduce the number of vertices we track visited information for, while minimally reducing the filtering rate. However, doing so implies a need also to store vertex IDs or log(#vertices) bits per entry (~6 bytes for a $2^{42}$ graph).

The orange line in Fig. 4a shows the total graph's % of vertices that must be retained in *each* LP if we retained information for all vertices that are visited from that LP. With 32 LPs, retaining all vertices visited from an LP using vertex ID would require storing 15% of the graph's total vertices or $2^{42} \times 0.15 \times 6$ bytes = 659GB *per LP*, exceeding the storage required for a bit-vector. If, instead, we assume an oracle that only stores the IDs of vertices accessed again in the future (blue line in Fig. 4a), storage can be reduced by 10×. We discuss two practical (i.e., non-oracular) approaches to reducing storage requirements for filtering purposes: (i) only retaining vertices within an individual superstep instead of throughout the entire execution, and (ii) selective storage by identifying "important" vertices (using in-degree as a heuristic).

### 3.3 Filtering within individual supersteps

In BFS, a frontier is created in each iteration, and messages are sent at once to all vertices in the frontier at the end of a superstep. In small-world scale-free graphs such as Kronecker-generated graphs, few supersteps end up having frontiers of the size of 30-40% of the graph, while the other supersteps are relatively small. Therefore, it is reasonable to expect that we have enough information locally within an individual superstep without retaining information about visits in previous supersteps. The orange line in Fig. 4b shows the filtering effectiveness if only the information from the current superstep was used. Due to the size of the supersteps, we see a minor drop in filtering effectiveness but up to a 3× reduction in storage requirements, getting much closer to the oracle filter (Fig. 4a). However, retaining all vertices even just within a single superstep still leads to several hundreds of GBs of storage required per LP (e.g., ~ 150 GB for a $2^{42}$ vertex graph with 32 LPs). While a simple optimization of removing the vertices with an in-degree of one reduces storage requirements by ~ 30%, we need additional heuristics to reduce storage requirements while retaining the vertices more likely to contribute to filtering.




(a) With naive filtering. (b) Filtering using information from current superstep. (c) Filtering with an in-degree threshold.

**Figure 4: Storage requirements for filtering.**

## 3.4 Filtering using in-degree threshold

Due to the power law nature of the graph, some vertices participate in more messages than others, based on their degree. We find that 80% of filtering actions are from 20% of vertices, which typically correspond to ones with high in-degree. Therefore, we consider using in-degree as a threshold to limit the size of the filter structures while maintaining high filtering effectiveness.

Fig. 4c shows a case of a $2^{27}$ graph on a system of 32 LPs, and a variable in-degree threshold on the x axis. The graph shows the ratio of the maximum storage and effective filtering rate to the maximum possible values. The maximum filtering rate is ~ 84% with the same storage requirement as keeping all vertices of the graph (~600GB). Due to the power law nature of the graph, the entries required (orange line) drop off exponentially compared to the filtering ratio. For example, 35% of max filtering is achieved with only ~0.6% of the maximum storage or ~ 4GB per LP. However, fixing a specific in-degree value threshold does not scale to larger graphs; instead, we can use vertex in-degree percentile as a harness to control the filter size and filtering rate for graphs of larger size. For $2^{27}$ vertices, the 99.9th percentile in-degree threshold corresponds to 16k entries that achieve 16.8% filtering. The required filter size increases linearly from $2^{20}$ to $2^{27}$, and the filtering rate increases by around 0.5% every time the graph's size doubles. With these, we project around 1GB storage required for ~23.5% filtering rate for a $2^{42}$ graph.

Although much smaller than the 100s of GBs required by naive filtering approaches, depending on the storage type and filter placement, these sizes still limit the types of practical hardware filtering units. For instance, GBs of storage will not fit in a typical switch's SRAM structures, but using an HBM to store the filtering information makes a few GBs viable.

## 4 IMPLEMENTATION CONSIDERATIONS

In hierarchical architectures, filtering can be performed at different levels. We investigate three cases.

*Software-only filtering in each group.* This approach significantly reduces the overall filtering effectiveness compared to what would be attainable if filtering information was aggregated across groups. As earlier indicated in Fig. 3, aggregating information across 8 groups can improve the filtering rate by 30%.

*Hardware-only filtering at the GC.* Filtering at the GC has the benefit of aggregation and reduced storage. The GC's filter would be larger, however, as it would need to retain vertices that have been visited from all of the GC's groups. For a $2^{42}$ graph, scaling linearly from results of a $2^{27}$ graph, we project 24% filtering rate with 1GB storage.

*Hierarchical filtering:* software-only filtering in each group *and* hardware filtering at the GC. A hierarchical filter would perform the same as hardware-only filtering at the GC, because every filter at a group contains a strict subset of vertices maintained at the GC. However, it will reduce traffic on the interconnect between the groups and the GC. Applying different thresholds for the filter at the group and GC levels would allow amortizing storage requirements across the two types of filters without affecting the filtering rate. Using the scaling methodology in Section 3.1, for an in-degree threshold of 99.5% percentile at the GC, we project that 42.5% filtering is attainable with (i) 10 GB at the group and 2.5 GB at the GC, (ii) 11.5 GB at the group and 1.5 GB at the GC, or (iii) 12.5 GB at the group and 250 MB at the GC.

## 5 CONCLUSION

Up to 99% of vertex visits in distributed BFS can be filtered with an oracle filter, and ~ 85% with a filter at the visit's source using node-local information. While filtering can be done in hardware or software, filtering at a higher level of a hierarchical architecture can aggregate information to achieve a higher filtering degree. We investigated several approaches to reduce the considerable storage requirements required for such filtering. Using in-degree as a criterion reduces the filtering rate but also reduces storage overheads significantly. Further reductions in storage requirements are possible through hierarchical filtering (first in software, then in hardware). Such a hybrid method is a promising approach to reduce communication overheads of BFS in hierarchical systems. A similar filtering approach can be extended to other key graph kernels beyond BFS, like triangle counting, but at a significantly higher cost in terms of storage and computational requirements.

**Acknowledgments.** This research was conducted under the umbrella of the IARPA AGILE research program (Army Research Office contract number W911NF22C0083). The views and conclusions contained herein are those of the authors.

## REFERENCES

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 2010. Scalable Graph Exploration on Multicore Processors. In *SC'10*.
[2] Scott Beamer. 2016. *Understanding and Improving Graph Algorithm Performance.* Ph. D. Dissertation. University of California, Berkeley.
[3] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC'12*.
[4] Aydin Buluç and Kamesh Madduri. 2011. Parallel Breadth-First Search on Distributed Memory Systems. In *SC'11*.
[5] Graph500. 2023. The Graph500 List. https://graph500.org.
[6] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *SC'05*.