Packed Memory Arrays for Dynamic Graphs in the Distributed Memory Setting

Rohit Agarwal
Alec James
Joshua You

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-235
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-235.html

November 20, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Thank you to Professor Aydin Buluc for suggesting this project and connecting us with Brian Wheatman and Helen Xu. Big thanks to the latter two for discussing our model with us, giving us serial code to get started, and providing us with the realistic graph workloads they used in their works. Finally, thank you to the CS267 staff for being helpful in office hours and teaching us so much this semester!

Packed Memory Arrays for Dynamic Graphs in the Distributed Memory Setting

Rohit Agarwal, Alec James, Joshua You

May 2023

Abstract

Sparse dynamic graph workloads are common for many modern database problems. Thus, it is necessary to have graph data structures that have high update and query throughput while also operating in a distributed manner, where many machines can access a shared structure. Furthermore, such a data structure should be cache-optimal in the face of locality-exploiting algorithms like Breadth-first search. The cache-friendly Packed Compact Sparse Row (PCSR) has already seen great use in the shared memory setting.

We propose an extension of the PCSR, the DistPCSR, which uses ideas from the serial one-machine Packed Memory Array (PMA) to build a routing layer between many PCSRs on different machines. This allows us to take queries while updates are still being resolved, and to share load between servers to stop any one from using too much memory. We provide some guarantees for this data structure, allowing one to be confident in its answers in a distributed setting. We also describe how to use the UPC++ framework's primitives to implement a DistPCSR.

Finally, we test our implemented DistPCSR on realistic graph workloads with algorithms that are friendly with PCSRs and share our results.

1 Introduction

In the real world, many sparse graphs are dynamically changing; this has been especially true in the age of the internet, where social network and internet graphs both lose and gain edges. As a response, there have been many works that investigate different data structures for this kind of graph processing. [6, 8, 10, 15]. These data structures must support a fast stream of updates (edge inserts/deletions) as well as a fast stream of queries. In this paper, we focus on the following setting. Suppose there are P servers, any of which can be queried about the status of any edge. We maximize the throughput of arbitrary sequences of queries and inserts. We want the guarantee of eventual consistency: if we break the inserts and queries into alternating stages and buffer between them, then we'd have exact correctness. We also do not care about deletions in this setting (though one can just view them as insertions with infinite edge weight since they are also sparse).

Beyond querying whether an edge is in the graph, we also seek to run common graph algorithms on these graphs, such as Breadth-first search (BFS) or PageRank (PR). In BFS, note that an important primitive is getting all of the neighbors of a particular vertex, which we can consider a range query. Thus, any data structures that can run these graph algorithms efficiently should use locality and store edges incident on the same vertex together to speed up these range queries. However, there is generally a trade-off between range query and update performance (e.g. dynamicity). In the static setting, Compressed Sparse Row (CSR) performs optimally with respect to cache misses due to storing edges in a contiguous sorted array. However, it is not great for the dynamic case, as it must move many items on every insert. Thus, Wheatman and Xu [16] concluded that replacing the edge array in a CSR with a Packed Memory Array (PMA) [2] to support polylogarithmic updates and range queries. The cache-locality of the contiguous memory in a PMA makes a PCSR a good pick in practice as well. Wheatman and Xu further parallelized [15] their data structure, with intricate locking schemes that allowed concurrent reads and writes to different parts of the PMA. However, they did this in the model with threads that had a shared memory.

Instead, we describe a **Distributed Packed Compressed Sparse Row** (DistPCSR), which is a distributed graph storage data structure supporting operation parallelism and dynamicity. In particular, each of the P servers has its own statically sized PCSR (set to the maximum memory that server can support) and is in charge of its own subrange (global set of edges). As the amount of elements in a single PCSR increases, the PCSR shares the edges with neighboring servers' PCSRs, changing which subrange each server is in charge of. Overall, we think that such redistribution is rare, and will amortize out in most settings (the worst-case input for a redistribute is when all of the PCSRs are evenly filled; but this is what we want in the first place).

As for actual implementation, many distributed data structures have been open-sourced, including the Berkeley Container Library [3]. In particular, we follow them and use UPC++ [1], a high-performance communication framework that allows one to use Partitioned Global Address Space (PGAS) programming. The reason we want to do this is it encapsulates commands like "redistribute" and "update subrange" into a remote procedure call (RPC) with a cleanly abstract interface.

First, we shall describe how the data structure works, providing a more precise version of the consistency guarantee we alluded to earlier. Then, we discuss the implementation of the data structure in UPC++ and share the throughput performance on realistic graph workloads. Finally, we discuss possible future directions and unexplored possible offshoots of PMAs.

2 DistPCSR Data Structure

2.1 PCSR Modifications

We modify the PCSR originally presented [16] (henceforth when we say "PCSR" we mean the one we define in the rest of this report).

We start with a **Packed Memory Array** (PMA) [2], which maintains elements in order in an array with spaces between groups of elements. For our purposes, these elements will be 64-bit unsigned integers. A PMA storing n elements supports inserts with amortized $O(\log^2 n)$ work (we will not consider deletions). Querying if a number is in the PMA take $O(\log n)$ work, and range queries $r(s, t)$ (which can compute any accumulator on the range) that use k elements have $O(\log n + k)$ work. They use $N = O(n)$ contiguous bytes of memory (some of which are empty) and PMAs have strict upper and lower bounds on their density, which are the amount of filled memory bytes divided by N. As more elements come in, the PMA will also naturally resize itself (which does not change the amortized time due to array doubling), but will also have a maximum capacity N_{max}, after which it is not allowed to resize anymore. The way this data structure works is there is an implicit binary tree with leaves of size $\log N$; during a search, one traverses the binary tree for the correct leaf, then looks in the leaf for the item. For ease of implementation, we also use the packed-left property discussed in section 3 of [15], wherein we enforce that all the nonempty elements in each leaf are in the left part of it.

Then, we form the **Packed Compressed Sparse Row** (PCSR) out of a single PMA. Vertices are represented as 32-bit unsigned integers. When we insert an edge (u, v), we concatenate u and v and insert the resulting 64-bit number into the PMA. We do an unpacking when reading. In this fashion, we preserve the guarantees of the PMA. Note that in such a setup, we do not store the mappings from vertices to the edges they control. There is no vertex set, as simply inserting an edge for a vertex that doesn't exist creates it. In fact, without looping through the entire PMA, it is impossible to know which vertices are even present in the graph. The reason we do this (and differ from prior treatment) is that when we distribute a vertex across (possibly many) PCSRs, it becomes more difficult to keep vertex pointers updated.

2.2 Initialization

On initializing a **Distributed Packed Compressed Sparse Row** (DistPCSR), each server s_i will initialize a PCSR with some maximum size related to how much memory it has (say it allocates N_{max} bytes). For a graph with vertex set V and $|V| < 2^{32}$, we take the entire space $[0, |V| \cdot 2^{32}]$ (the space of all edges, by our representation above) and divide it evenly into P parts, and each processor is said to control a subrange of items $\left[i * \frac{|V| \cdot 2^{32}}{P}, (i+1) * \frac{|V| \cdot 2^{32}}{P}\right)$; server s_i stores this information locally in an array called $\texttt{ranges}_i$. In the other parts of this data structure, the information may be stale; we will have tricks to work around this. The distribution can be thought of as a 1-dimensional distribution, but parts of multiple rows of the global CSR could fit inside one local PCSR.

2.3 Insert and Query

We use the range data to implement the subroutine `target_rank`, which routes an edge to the server we think has the information. We will see that because of the way we manage subranges, this information does not have to be perfect.

```
def target_rank(e):
  return r such that ranges[r] <= e && ranges[r + 1] > e
```

Here is the pseudocode for an insert, where we define `redistribute` in the next section.

```
def insert(e):
  if (target_rank(e) == self.rank):
    if self.pcsr would fill up:
      redistribute()
    insert e into self.pcsr
  else:
    forward the request to target_rank(e)
```

Querying an edge has the same logic, but it does not check for fullness and instead returns the Boolean back from the request in the last line. Finally, querying all the neighbors of a single vertex is also similar, except one gets all the edges starting from $(v, 0)$ and ending at $(v, 2^{32} - 1)$. There may be multiple target ranks, so we send requests to all of them before formulating a response.

A particular optimization for inserts was to batch them into groups of approximately 4×10^3. Higher batch sizes were experimentally shown to achieve better performance, at the cost of the data structure not being as frequently up to date. However, we also placed value on freshness of the data within the PCSR, thus did not push the batch size higher. A rank would achieve batched inserts by, when receiving an edge, grouping it into one of n buffers reserved for the edge's target rank. It would then periodically flush this set of n buffers either when full, or when a graph traversal is initiated.

2.4 Redistribution: Fixed VS Active

We formulate two different implementations of the redistribute function, fixed and active.

In the **fixed** (sometimes called static, but still dynamic updates) subranges case, the `ranges` stay fixed for every server, and we just raise an exception if a PMA gets too full. In the dynamic case data-structurely, this actually seems to work fine; even with high-degree vertices, the vertices next to them in number are rarely big enough to overwhelm a subrange. If you do run into an issue, a random permutation of the labels fixes the problem. Nevertheless, it will fail if your graph has vertices with very very large vertices or a decently-sized clique.

In the **active** (dynamic) subranges case, we initialize `ranges` the same as above, but allow `ranges` to change in order to account for PCSRs filling up. Suppose s_i initiates the redistribute. The algorithm proceeds in a few stages.

1. **Locking** We require that only one redistribute occurs globally throughout the data structure. Our reasoning is that having multiple going at the same time adds unnecessary complexity, while redistributes are rare enough that this isn't a significant slowdown. In addition, each server has a `redistributing` flag. If set to true, then it cannot be inserted into, since it's redistributing, so inserts will be buffered instead. Thus, at the beginning, we acquire a lock, releasing it if in the process of acquiring, s_i already went through a redistribute while acquiring (i.e. someone else redistributed with us).

2. **Hierarchical Search** Here we get a lot of inspiration from the redistribution process in PMAs. Let S, the number of servers, be a power of 2; the servers can then be seen as leaves in a perfect binary tree. When a server s_i wishes to redistribute, it first checks in with its neighbor in the 2-level subtree containing i. If both together are too full, then we go one subtree higher. We keep going until we are no longer too full; if we reach the top of the tree, then there isn't enough space over all the servers for the graph, and there is no hope. During this process, those servers that will participate in the redistribute (call this set R) also have their `redistributing` flag set to true.

3. **Issue Redistribute Commands** Once we figure out which servers s_i is redistributing with, s_i issues redistribute commands to each server. A redistribute command is a tuple (a, b, k) which tells the server s_j who receives it to request indices $[a, b)$ from server s_k. Upon receiving a redistribute command, the server immediately requests the proper indices and stores them in a `temp` array. During this process, s_i updates $\texttt{ranges}_i$ to the new subranges that the items in R control. There is a barrier for all servers to finish this step, so we do not start changing arrays until all requests are done.

4. **Swap Arrays** All the processors set the `temp` array as the new PMA (running PMA-level `redistribute` to make sure their density is correct). During this process, the servers in $R \setminus \{s_i\}$ get updates of the subranges from all other servers. We again wait for all of these to finish. Once a server is done swapping (which isn't s_i), its `redistributing` flag can be set to false.

5. **Subrange Synchronization** We then send the new subranges off to all other servers that did not participate in the redistribute. These subranges are timestamped, and a server updates its own subrange only if the timestamp of the stored subrange is older than the incoming one. We do not wait for this to finish. Once the requests are sent, we set our `redistributing` flag to false and release the lock on global redistributing s_i had acquired.

2.5 Correctness

In the fixed case, correctness is fairly trivial. In the active case, we claim the following guarantees about our data structure that gives it reasonable correctness and consistency.

Guarantee 1. *At any point in time other than between steps 3 and 4 of a* `redistribute`*, a server s_i's own* $\texttt{ranges}_i[i]$ *is consistent with the data in its PMA. Furthermore, the* $\texttt{ranges}_i[i]$*'s form a disjoint cover for the entire range.*

Proof. Note that the ranges satisfy the disjoint cover property initially. Any update to s_i about its own subranges must come from someone it is redistributing with. But once it finishes step 4, its local PMA contains all the elements in its subrange (from `temp`). Thus, its data is consistent. Furthermore, within a redistribute, the subranges are reassigned so the subrange controlled in total by the servers in R is split up into a disjoint cover, but for $s_k \notin R$, nothing is changed. Thus, we must maintain this disjoint cover property. ☐

This means that every inserted element belongs in exactly one globally unique $\texttt{ranges}_i[i]$, since we do not do inserts while a redistribute is going on. This means we can now be assured that all edges are accounted for.

Guarantee 2. *The data structure does not drop, duplicate, or infinitely route any edges (assuming a finite amount of total operations). That is, no matter the order of requests (messages), all inserted elements will eventually be inserted into its globally unique server (at the time of being added to a local PMA).*

Proof. Since an edge is only moved if it cannot be inserted locally, it definitely cannot be duplicated. Furthermore, due to forwarding, an edge cannot be dropped if it is out of range. Finally, suppose that the edge e is sent from server s_i to server s_{j_1} but $\texttt{ranges}_i[j_1]$ was outdated and no longer contains e. For this to be the case, s_{j_1} must have redistributed with another server s_{j_2} after $\texttt{ranges}_i[j_1]$ was updated and thus has a more recent subrange associated with it. We can have this happen inductively; since there are finitely many redistributes and each subrange is more recent, we will eventually get to the most recent relevant redistribute and find a correct subrange, making the insertion terminate. ☐

Finally, we give some convergence guarantees that establish the notion of synchronization we'd like to have across the servers.

Guarantee 3. *Consider a set of operations given in alternating insert and query phases. During an insert phase, we allow all the servers to fully finish all inserts and redistributes (but not necessarily update all subranges) before the next query phase. Then, all queries reflect the status of edges described by the insert phases before them.*

Proof. Since there are finite phases, guarantee 2 holds, and thus every inserted edge exists in some PMA. This means we have correctness as long as queries are routed exactly once to the correct PMA. By the same forwarding argument as in guarantee 2, a query will eventually find its way, even when subranges are not updated. ☐

2.6 UPC++ Implementation

We implement the "requests" described above as remote procedure calls (RPCs) in UPC++, returning futures that we can wait on or batch together. In addition, during any "local insert," we instead append to a request queue of inserts, which acts as our incoming buffer. Furthermore, every time we run step 5 of the active subranges redistribute, we also keep track of how many RPCs are active; whenever we `upcxx::barrier`, we also make sure that these counts are kept to zero. Whenever we make `upcxx::progress()` and spend computation on futures, we also start flushing the queue manually. We also batch inserts, which means that we accumulate many edges that are to be sent to some node in a list and send the list when it gets to a large constant size, which reduces the number of messages. See section 3 for a more in-depth discussion of batch sizing.

3 Experiments

3.1 Experimental Setup and Algorithms

We implemented parallel BFS as discussed in [4] and a parallel PR inspired by the parallel BFS algorithm. PR was implemented with an early stopping criteria similar to that of the Ligra library, determined by the ℓ^1 norm between the two most recent PR steps. Upon falling below $\epsilon = 10^{-7}$, the algorithm terminates, the same stopping criterion as used for Ligra. Both were implemented using the bulk synchronous parallel model, wherein the computation takes place in phases, with explicit synchronization delineating the different phases.

For the graph workloads, we used rMAT [5] and LiveJournal graphs, two of the types of graphs studied in the PPCSR work [15].

We ran our benchmarks on the Perlmutter supercomputer by NERSC, using the CPU nodes which contain 2 AMD Epyc 7763 (Milan) processors with 64 physical cores each. The number of nodes allocated to a run ranged between 1 and 8 inclusive, and were ran with 64 ranks per node (using more exhausted the memory capacity due to the shared segment and UPC++'s internal buffers for send and receives having a fixed overhead per rank).

3.2 Results

Our performance numbers are reported on the following pages. We compare several settings of our data structure against a shared-memory, non-dynamic graph processing system to act as a reference and an upper bound on our performance (extrapolating Ligra's performance to beyond 64 ranks, a physically implausible scenario due to its implementation in shared memory).

We display performance comparisons between our baseline model – a statically allocated PMA with no inter-processor redistribution, 1D source-vertex-based routing, and waiting on inserts – and our load balanced model, which allows resizing, inter-processor redistributes, and dynamic ranges based on both source and destination vertices on the inserted edges. We also compare our performance of these models to a roughly calculated upper bound described in each figure.

While the insert and query throughput of our data structure appears to scale with the number of nodes, the speed of BFS and PageRank decreases after a certain point. This can probably be explained by the fact that our data structure is doing insufficient computation over which the latency of the required all-to-all communications phases in both algorithms can be amortized. In particular, since the problem size stays fixed, the proportion of edges in the graph that you must communicate for increases, decreasing the use of the "fast path" we implemented to remove redundant RPCs to your own rank.

Essentially, the computational intensity of BFS is almost nonexistent, meaning we are entirely memory bound. The computational intensity of PageRank is also very small, since at its heart it can be formulated as a power iteration on a (slightly modified) stochastic transition matrix derived from the adjacency matrix, requiring repeated matrix-vector multiplies (BLAS-2) with low computational intensity. This is made worse by the fact that we do not do any of the graph-partitioning algorithms to reduce communication, so many of the "memory accesses" require expensive RPCs onto other processors. We wanted to focus on implementing dynamicity and did not carefully optimize the "applications" running on top of our distributed data structure, but this could potentially have helped our performance. In either case, our program is basically running at the bottom of the roofline, a roofline whose slope is dictated by the speed of the network. This explains the poor performance of our code in comparison to Ligra, a purely shared memory implementation.

Overall, the best metric for our novel data structure is simply raw insert throughput. Our inserts, which are not "bulk" inserts in which the data structure is modified with multiple new edges in a single operation (which amortizes the cost of relatively cache-inefficient binary searches and allows for a more efficient "merging" strategy to be used), compare favorably to the pointwise inserts of other data structures reported in the PPCSR paper.

Insert throughput	T_{PPCSR}	T_{Aspen}	T_{DistPCSR}
Live Journal	6.31×10^5	8.07×10^4	8.91×10^6

We also explore the performance effects of batching inserts together to reduce the number of RPC calls. Note that this is different from the "bulk inserts" discussed earlier; while many inserts are bundled together, each one incurs the cost of a binary search to find the right location and the cost of sliding several elements to make room for the new one. Batch sizes appear to have a relatively minor effect, up until the point that they are so large as to be able to hold the entire dataset in buffers for sending. At this point, all of the updates to the graph are deferred to the end, and the entire graph is populated after one enormous all-to-all; with this change in behavior, the insert throughput approximately doubles in magnitude since there is much less load imbalance. However, this is not a

very realistic situation, and our experiments were run with a much more modest batch size (2^12) to simulate a more realistic scenario of online updating on a very large graph.

One important hyperparameter of our data structure is its load factor (LF), or the fraction of the data structure reserved for modification. At lower values, leaves are less full, which means that fewer elements must be shifted over to make room for new elements, which doesn't change the asymptotic behavior but does improve the insertion throughput, especially since it means fewer expensive multi-rank redistribution operations are required. However, it is also much less memory (and cache) efficient.

Another important performance consideration which has not been given much treatment in the literature surrounding PMAs is the importance of the density bounds on performance. Initially, our implementation ignored the density bound, which was causing many needless redistributes and hurting performance. However, imposing too aggressive of density bounds is also problematic, as it makes the data structure too sparse. We changed the density bounds to decrease with each level in the local PMA by 0.01 and decrease for each level of the routing layer by 0.02. Tuning these density bounds is not something we had time to do for this work, but is a potential avenue for exploration in the future. In particular, some method of automatically tuning this hyperparameter can be considered.

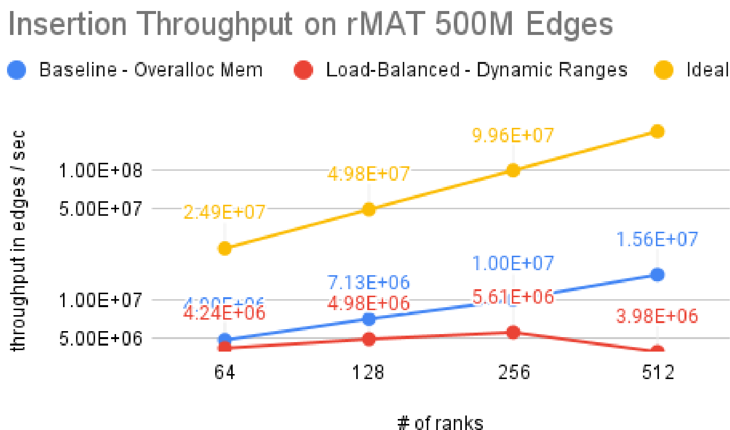

Figure 1: Ideal throughput was calculated by taking the average insertion speed of a single edge among 10 inserts, taking the inverse, and then multiplying by the total number of ranks. It is worth noting that though the baseline has higher throughput in this graph, when dynamicity was enabled for each processor's PMA it was much slower, and could not finish inserting the whole graph within a reasonable amount of time. This exhibits clearly that the redistributions and resizes hinder throughput.

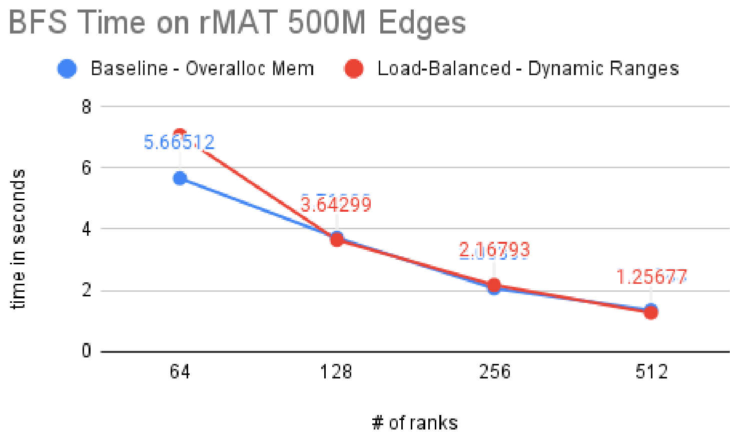

Figure 2: We compare the same two models on BFS. Parallel BFS on the baseline model has the advantage of having nodes partitioned solely along source vertices, which gives it the advantage of always having access to the neighboring edges locally. However, the load-balanced model has the possibility to split a single source vertex among several ranks, which might add an extra communication step when computing neighbors.

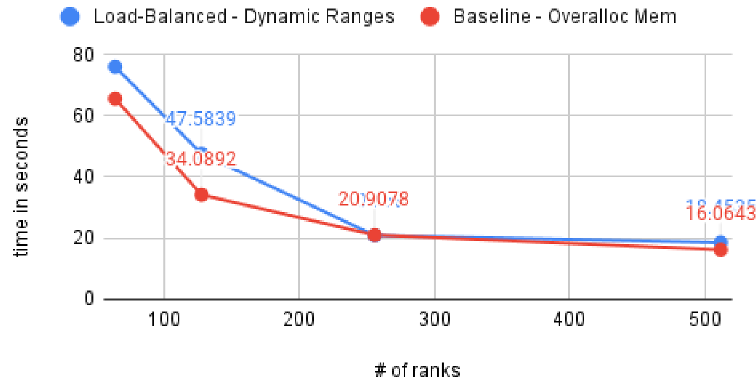

Figure 3: PageRank runs significantly faster on Baseline until 256 ranks, where the runtime is approximately the same. Most likely due to better neighbor locality as described in 2.

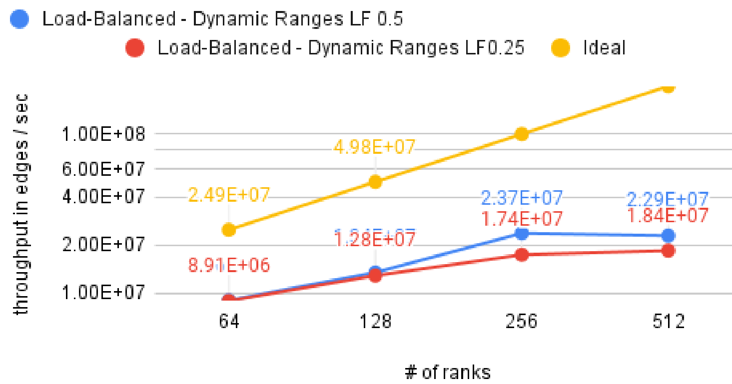

Figure 4: With a lower load factor we get slightly less insertion throughput. A possible explanation for this is that the LiveJournal graph is not very big on its own, which means redistributes are rare in both LF=0.5 and LF=0.25. This means that the cons of allocating a lot of initial space are more prevalent. Since data is more sparsely distributed among the lower load factor, this means that there are most likely more cache misses due to all the wasted space.

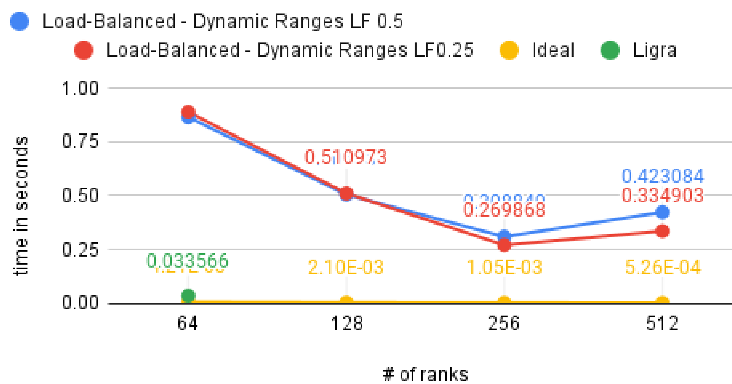

Figure 5: BFS with the LF=0.25 seems slightly faster than LF=0.5, though still far from ideal. This difference is not statistically significant though, and can be attributed to variance in Perlmutter nodes. Ideal throughput is calculated as described in 1.

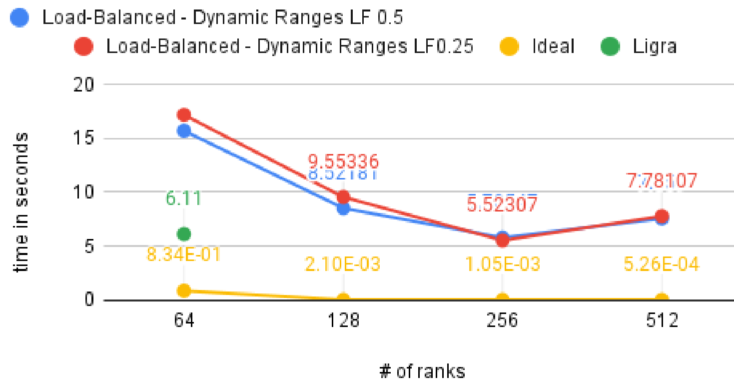

Figure 6: PageRank performance between the various load factors has almost no difference. Ideal runtime here is calculated by taking the PR runtime of Ligra on 1 node and dividing by the number of ranks.

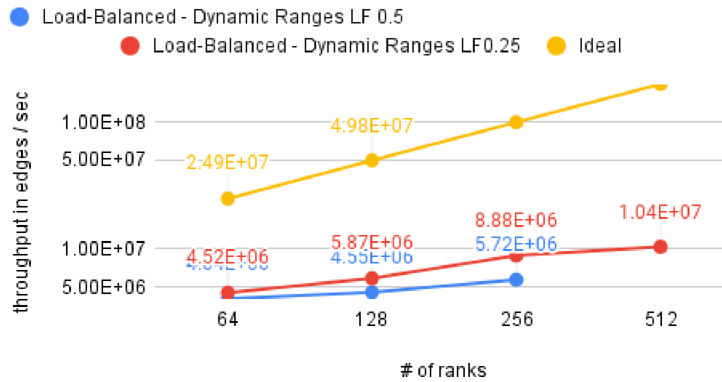

Figure 7: On the much larger RMAT graph, the throughput of the load factor is much higher, due to fewer redistributes and resizes. Communication costs of routing and redistributes in both settings keep both lines far below ideal.

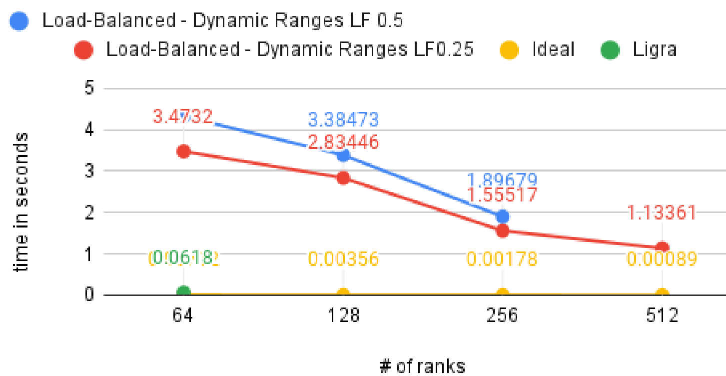

Figure 8: BFS times are slightly faster with lower load factors, and would completely time out with 512 nodes on LF=0.5.

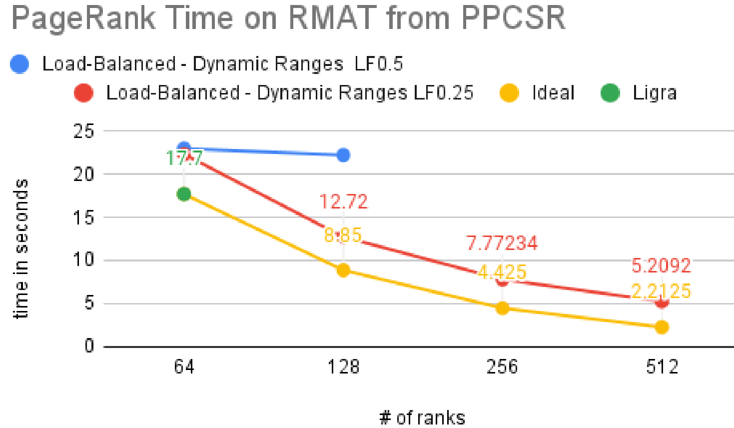

Figure 9: PageRank times with LF=0.25 are significantly faster than with LF=0.5. LF=0.5 times out when ran with 256 and 512 ranks. Ideal runtime here is calculated by taking the PR runtime of Ligra on 64 nodes and then dividing it by the scale factor of the number of ranks.

4 Related Work/Possible Extensions

First, we give a brief review of data structures that also attempt work in the distributed graph streaming setting. Many of them focus on the different but also relevant problem of graph partitioning [9, 17]. For the case of actually maintaining the entire graph, the common data structural pattern in [18, 19] is to use a push-based approach, where, similar to our work, we give different servers different updates and is their responsibilities to push this information to all other servers. However, there is some merit to the idea of a pull-based approach [7, 11] instead, where each server pulls in updates from their neighbors every so often. The problem with such a notion is that you must have a redundant version of the graph for it to work and thus more redundant computation, even when the graph is small. Ligra [12] uses a mixed push-pull scheme in the shared memory setting, based on graph processing size.

More immediately, another obvious improvement would be implementing a fine-grained locking system so that the single global redistribute lock is not a serial bottleneck. For our experiments, the contention over the lock was relatively minor, since the number of ranks we were running our code on is small. However, with more ranks, this would quickly become an issue that prevents further scaling. To accomplish this, we would most likely need to implement a system similar to that used in the PPCSR paper [15], which details an elaborate scheme to prevent deadlock between multiple threads attempting to redistribute at once.

We also consider the possibility of some kind of 2-dimensional distribution, wherein each server is in charge of a block of the adjacency matrix, rather than a segment of the vectorized adjacency matrix as we do. In static sparse matrix-vector applications (such as PageRank) a 2-dimensional distribution has been shown to be better in terms of overlapping computation and communication [13] as well as graph (block) reordering, so we wonder if those kinds of techniques could also be employed in the dynamic setting, where they are a lot harder.

Finally, we could also consider using other data structures for the bottom indirection layer (the local sets). Some candidates could be B+-trees, a sparse PMA [14], or even regular adjacency lists. These would have different trade-offs in insert versus query versus BFS. An interesting idea would be using a PPCSR as the bottom layer, since UPC++ overhead for RPCs is quite significant. In particular, while communications between ranks in the same node don't incur serde overhead, they are still implemented as message-passing rather than purely shared memory operations, which leaves some performance on the table.

5 Acknowledgements

Thank you to Professor Aydin Buluç for suggesting this project and connecting us with Brian Wheatman and Helen Xu. Big thanks to the latter two for discussing our model with us, giving us serial code to get started, and providing us with the realistic graph workloads they used in their works. Finally, thank you to the CS267 staff for being helpful in office hours and teaching us so much this semester!

6 Code

Our code is available publicly online at https://github.com/girantinas/267-pma-project. The code for DistPCSR is in `dist_pcsr.hpp`. Its driver code is `main_dist_pcsr.cpp` and rMAT graphs can be generated using `rmat.cpp`.

References

[1] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. Upc++: A high-performance communication framework for asynchronous computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973, 2019.

[2] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[3] Benjamin Brock, Aydin Buluç, and Katherine A. Yelick. BCL: A cross-platform distributed container library. *CoRR*, abs/1810.13029, 2018.

[4] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. volume 6, 04 2004.

[6] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012.

[7] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[8] Oded Green and David A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.

[9] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 439–450, 2019.

[10] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[11] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146, New York, NY, USA, 2010.

[12] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[14] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu. *Optimizing Search Layouts in Packed Memory Arrays*, pages 148–161.

[15] Brian Wheatman and Helen Xu. *A Parallel Packed Memory Array to Store Dynamic Graphs*, pages 31–45.

[16] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. pages 1–7, 09 2018.

[17] Wei Zhang, Yong Chen, and Dong Dai. Akin : A streaming graph partitioning algorithm for distributed graph storage systems. 05 2018.

[18] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: A distributed framework for prioritizing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 24(9):1884–1893, 2013.

[19] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *CoRR*, abs/1710.05785, 2017.