Constant-Memory Graph Coarsening

George M. Slota
Rensselaer Polytechnic Institute
slotag@rpi.edu

Christopher Brissette
Rensselaer Polytechnic Institute
brissc@rpi.edu

***Abstract*—Graph coarsening is an important step for many multi-level algorithms, most notably graph partitioning. However, such methods often utilize an iterative approach, where a new coarser graph representation is explicitly constructed and retained in memory at each level of coarsening. These overheads can be prohibitive for processing massive datasets or in constrained-memory environments like GPUs. We develop a data structure (CM-Graph) for representing coarsened graphs, which can be used with any adjacency-based graph representation. The CM-Graph data structure uses a constant amount of memory, regardless of the desired level of coarsening. In addition, CM-Graph does not require modification to the existing graph representation, it offers a several-fold memory savings in practice, and it can even accelerate graph coarsening, due to not having to explicitly construct coarser graph structures. We further describe efficient GPU parallelizations of the CM-Graph subroutines for adjacency access, which can also be utilized in most arbitrary graph computations without modification.**

***Index Terms*—graph coarsening, graph partitioning, parallel algorithms**

I. Introduction

This paper considers the problem of graph coarsening with a specific focus on its usage within a graph partitioning-like framework. Graph coarsening can be described as the process of taking some input graph G (or matrix) and producing a smaller output G' that closely represents G per some objective(s) [1]. The primary motivation for coarsening is to accelerate a computationally difficult problem on a given input. Coarsening has found wide applications in graph partitioning [2, 3] and clustering [4], graph neural networks [5], graph embedding [6], and linear algebraic applications [7, 8, 9], among a plethora of others [10].

The vast majority of prior work in graph coarsening has focused on how to optimize the coarsening procedure to produce some G' that meets the desired objectives for a given application. Generally, these optimizations have focused on computational time to solution [3], parallelization efficiency [2, 11, 12], and quality of solution per the given objective [13, 14, 15]. Many coarsening procedures, particularly in graph partitioning [16, 17], follow a basic multi-level approach, where the graph G is iteratively coarsened until the resulting G' is small enough that the target computational problem becomes tractable. A solution on the coarsened graph is then extrapolated back to the original input.

For our primary motivating problem, graph partitioning, such a procedure produces many intermediate G_i graphs that are maintained in memory to iteratively *refine* the produced solution on G' as it is extrapolated back to the original G. This requires the explicit creation and storage of all intermediate graphs in memory or disk. State-of-the-art coarsening implementations often spend more than half of their processing time during graph construction [2]. For massive graphs or when processing in limited memory devices (e.g., GPUs), these overheads can be prohibitive to scalability.

Contributions: We introduce CM-Graph and describe its implementation and optimizations for serial and shared memory parallel applications on CPU and GPU. We demonstrate it under two representative coarsening algorithms, including greedy/random coarsening and heavy edge coarsening. These two algorithms represent the primary ways the adjacencies of a vertex are processed: basic adjacency reachability/traversal and computing some reduction (e.g., `max`, `sum`) over the vertex's entire adjacency list. Our code is available on GitHub[1].

II. CM-Graph: Constant-Memory Coarsening

A. Basic Graph Coarsening

We first consider a graph $G = (V, E, W_v, W_e)$, where V ($n = |V|$) defines a set of vertices, E ($m = |E|$) defines a set of edges, and W_v and W_e are weights for vertices and edges (with possibly multiple weights per each). This G can be coarsened in many different ways [10], though we focus on *pairwise aggregation* in this work, or the processes of *contracting* edges. An edge contraction combines both endpoint vertices into a super-vertex, where the super-vertex contains all adjacencies of the original vertex and the summed vertex weights. When multi-edges appear during contraction, they can be combined into a single edge with the summed edge weights. Vertices can also be contracted in instances where no direct edge exists between them (e.g., 2-hop coarsening) and it is also possible to contract three or more vertices at once.

For many coarsening algorithms, a maximal *matching* is first computed. A graph matching is a set of edges that share no endpoint vertices, and it provides a set of independent edge contractions. To coarsen G, matched edges are contracted and the endpoint vertices are merged into super-vertices. This process is then iteratively repeated until G' is sufficiently coarse, with all graphs at the intermediate coarsening levels retained in some array. Note that the literature is quite broad for graph coarsening, and there are plenty of methods which do not fit this basic outline. In this work, we focus on the coarsening algorithms that fit this outline for simplicity, though our methods are much more broadly applicable.

[1] https://github.com/HPCGraphAnalysis/CMGraph

B. CM-Graph Coarsening Data Structure

We will describe CM-Graph and its algorithms in a relatively low level. To begin, we assume we have some graph G with n vertices and m edges stored in a compressed sparse row (CSR) format as $G = (adjArray, offsets, vertexWeights, edgeWeights)$. We use the term *vertex ID* or vertex identifier to denote some numeric value that uniquely identifies some vertex u, typically in $[0 \ldots n-1]$. A CSR utilizes two arrays to access edges (or *adjacencies*) of u. An integer array of length $2m$ ($adjArray$) contains all adjacencies for all vertices in some order, usually by vertex ID. A second array of length $n+1$ ($offsets$) contains the start indicies in $adjArray$ for all vertices' adjacencies. E.g., $adjArray[offsets[u]]$ is the first adjacency of u and $offsets[u+1] - offsets[u]$ gives the degree of u.

Our coarsening scheme is centered on the ancillary CM-Graph data structure, defined in Table I. We require no modification to the original graph CSR representation (or any other adjacency-based representation) to use CM-Graph. Our scheme is centered around linking the adjacency lists of vertices during a contraction (or merge) into a super-vertex. This allows us to access these linked adjacency lists from a super-vertex during coarsening algorithm processing. Five primary arrays of length n to track graph coarsening, with an additional n-length array and integer used to accelerate processing during multilevel processing. The final row in Table I is not part of the CM-Graph data structure itself, but gives a (very very loose) worst-case bound for memory overhead when accessing a coarsened adjacency structure.

Name	Size	Brief Description
Head	n	Head[u] is the super-vertex that has consumed u
Next	n	Next[u] is the next vertex after u in the adjacency chain
Level	n	Level[u] is the coarsening level that u has merged
Degree	n	Degree[u] is the sum degree of super-vertex u's children
vWeight	$n(w)$	vWeight[u] is the sum weight of super-vertex u's children
Supers	n	Array containing all current super-vertices
nSuper	1	Number of current super-vertices
ExpAdjs	$O(m)$	Worst-case overhead for parallel adjacency expansion

TABLE I
THE NECESSARY DATA CONTAINED WITHIN CM-GRAPH. 'NAME' INDICATES HOW IT WILL BE REFERENCED WITHIN THE TEXT OF THIS PAPER AND 'SIZE' IS THE LENGTH OF THE NAMED ARRAY.

As can be seen in Table I, for each vertex u in G, we store its 'Head', which is the current super-vertex that u has merged into. The array 'Next' is how we keep track of the linked list of adjacencies from the super-vertex. The value of Next[u] gives the next vertex in the linked adjacency list of super-vertex Head[u]. Initially, both Head[u] and Next[u] are simply set to u. 'Level' gives the coarsening level (or more generally, an order) at which vertex u has merged into some other super-vertex, initially set to zero. Note that a vertex can merge into some other super-vertex only once. 'Degree' gives the sum degree of all children of some super-vertex u, where the children are all vertices that are part of the linked adjacency and have their Head set to u. It is initialized to the original degree of u. While it is not explicitly necessary to track this, it is useful for the adjacency expansion algorithms described later. Similarly, we use 'vWeight' to track the sum vertex weights. This is not necessary for unweighted graphs and it would be a multi-dimensional array for a graph with each vertex having w weights. Finally, 'Supers' holds the super-vertex IDs for the current coarsening level and 'nSuper' gives the number of super-vertices. These again are not strictly necessary, though it helps accelerate processing, particularly at higher coarsening levels.

C. Merging

Consider a merge of vertex v into vertex u in level k, visualized in Figure 1. In the prior level $k-1$, both u and v are super-vertices. After this current merge, v will be absorbed into super-vertex u. To perform this update, vertex v sets it Head to u and its Level to k. Vertex v and all of its children set their current Head to Vertex u, while the last prior merged vertex of u updates its Next value to v. This is the most expensive part of merging, as it is required to traverse the linked list of Next vertices to update the prior 'tail' (z as given in the figure) and perform all needed updates to the Heads values. An alternative approach is to instead not store Heads and compute them for each vertex in place during processing. However, as a vertex can appear multiple times when expanding linked adjacency lists (discussed next), the extra computational cost is considerably higher with only modest memory savings.

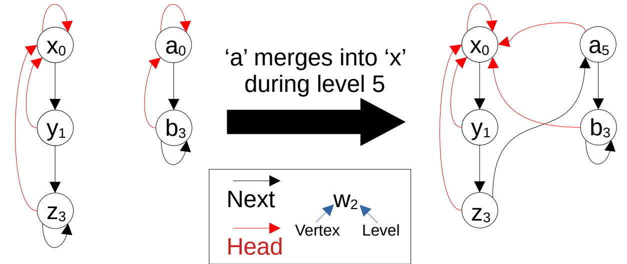

Fig. 1. A visual example of how a merge of vertex a into vertex x would occur. Vertex a and b update their new head to super-vertex x, previous tail vertex z now has its Next set to x, and x updates its Level to the current coarsening level. In addition, the Degree and vWeight(s) of super-vertex x would also be updated to now include the Degree and vWeight(s) of prior super-vertex a.

To unmerge vertices by one level, the above procedure is reversed. Any vertex u with the Level one lower than the current is identified as a new super-vertex, the super-vertex Head[u] traverses its linked adjacencies to compute a new Degree and vWeight(s) and 'cut' its linked adjacencies where they point to u by updating the tail vertex's Next value, and u and u's children update their Head values to u, and u updates its Level back to 0.

D. Accessing Adjacencies

We consider two primary means to access adjacencies, depending on the algorithmic use case. Consider accessing adjacencies of super-vertex u. One method is to access its adjacencies and all the adjacencies of its children, directly as a linked list. We access the adjacencies of u, and then

iteratively access the adjacencies of $curVert = \text{Next}[u]$ while $\text{Next}[curVert] \neq curVert$. See Algorithm 1, where we traverse a graph G's CSR adjacency list from some super-vertex u. This approach is used for a greedy coarsening algorithm, where we select the first vertex that has not been marked for a merge on the current coarsening level.

Algorithm 1 Basic Access of Adjacencies of Super-vertex u

1: **Input:** CSR Graph G, CM-Graph C, Vertex u
2: $adjIdx = G.offsets[u]$
3: $curVert = u$
4: **for all** i in $1 \dots C.Degree[u]$ **do**
5: $\quad a = \text{GetNextAdj}(G, C, curVert, adjIdx)$
6: $\quad v = C.Head[a]$
7: $\quad$ Do processing given edge (u, v)
8: **return**
9: **procedure** GETNEXTADJ($G, C, curVert, adjIdx$)
10: $\quad v = G.adjList[adjIdx]$
11: $\quad$ **if** $adjIdx + 1 \geq G.offsets[curVert + 1]$ **then**
12: $\qquad curVert = C.Next[curVert]$
13: $\qquad adjIdx = G.offsets[curVert]$
14: $\quad$ **else**
15: $\qquad adjIdx = adjIdx + 1$
16: $\quad$ **return** v

The second method explicitly creates the weighted adjacency list of u for the current level, given in Algorithm 2. To do this, we need to access all adjacencies as described above, tracking all unique super-vertices v and summing edge weights for each v to get the effective edge weight between u and each v. This procedure can be easily accomplished via a hash table, and we use an efficient serial implementation from prior work for this purpose [18]. Note that in Algorithm 2, the *Map*.InsertOrUpdate(v, w) function will place key-value pair (v, w) into the table if key v does not yet exist; otherwise, w is added to the current value for v in the table. We also note that this procedure will necessarily add memory overheads while coarsening or processing using CM-Graph, and we discuss this drawback later in our analysis section.

Algorithm 2 Hash-based Adjacency Expansion

1: **Input:** CSR Graph G, CM-Graph C, Vertex u
2: *Map* = HashTable()
3: $adjIdx = G.offsets[u]$
4: $curVert = u$
5: **for all** i in $1 \dots C.Degree[u]$ **do**
6: $\quad w = G.EdgeWeights[adjIdx]$
7: $\quad a = \text{GetNextAdj}(G, C, curVert, adjIdx)$
8: $\quad v = C.Head[a]$
9: $\quad$ *Map*.InsertOrUpdate(v, w)
10: **return** $\{$*Map*.keys(), *Map*.values()$\}$

Extracting all unique key-value pairs gives us lists of adjacencies (keys) and edge weights (values) for vertex u. We note that this second procedure can be used in any arbitrary coarsening or graph algorithm, given that the vast majority of graph processing algorithms perform computations by accessing the adjacencies of vertices.

E. Creating a k-level Graph

Given the coarsening of the graph to K levels, we can also explicitly construct a graph structure at any arbitrary level $k \leq K$. To access the adjacencies of u at some arbitrary level k, we examine the Level$[v]$ values for each $v = C.Next[u]$ vertex encountered during the above procedures. We stop the linked list traversal when a Level$[v]$ of $k + 1$ is found. To construct a full graph representation, we first identify our set of Heads as vertices u with Level$[u] \geq k$, indicating that they were super-head vertices until level k. We can then access their adjacencies for level k as described above, and use those adjacencies to fill an edge list or explicitly construct a new CSR or similar graph representation.

F. Parallelization

A straightforward CPU parallelization of a matching-based coarsening algorithm would assign in parallel every vertex in a given level to some thread. Each thread then attempts to identify a match for their owned vertex (with some synchronization), with all identified matches across all threads being merged in a later step. For such a scheme, our method can be directly implemented without modification to the underlying algorithm, as all expanded head adjacencies can be processed independently and a proper matching will not have any conflicts during merging. For the merging of more than two vertices outside of an explicit matching, a bit more synchronization is required, though the idea is the same. We implement such a naive coarse-grained parallelization using OpenMP for a comparison baseline.

However, for GPUs, a more fine-grained approach is generally required for performance on real-world graphs with an irregular degree distribution [19, 20, 21]. Modern GPU processing of such graphs considers parallelization across edges, where the adjacencies of u are processed in parallel by a warp, thread block, or some other construct of multiple threads.

Hence, we consider two more fine-grained parallelizations of our linked adjacency list expansion. With a super-vertex's adjacencies expanded and reduced into arrays, an adjacency vertex$\rightarrow$thread mapping and processing can proceed as normal. The first parallelization is relatively straightforward: we simply parallelize Algorithm 2 by using a thread-safe hash table, optimized for GPU from prior work [22], parallelizing over the insertion loop.

The primary phase of this approach, given in Algorithm 3, is for each thread to discover their assigned adjacencies of super-vertex u by traversing u's linked adjacency list. To do this, a thread initially computes their effective offset $tOffset$ from u's offset in the CSR $G.offset$ array as a function of their thread ID $threadIdx$. When their offset is greater than the number of original adjacencies for u, it updates their current vertex $tVert$ to the next one in the linked adjacency list and

Algorithm 3 GPU Hash Parallelization of Adj. Expansion

```
1: Input: CSR Graph G, CM-Graph C, Vertex u
2: Map = ThreadSafeHashTable()
3: tOffset = threadIdx
4: tOffsetSum = tOffset
5: tIter = C.degree[u]/blockDim + 1
6: tVert = u
7: while tOffsetSum < C.degree[u] do
8:     while G.offsets[tVert] + tOffset ≥ G.offsets[tVert+1] do
9:         tOffset −= (G.offsets[tVert + 1] − G.offsets[tVert])
10:        tVert = C.Next[u]
11:    w = G.edgeWeights[G.offsets[tVert] + tOffset]
12:    a = G.adjList[G.offsets[tVert] + tOffset]
13:    v = C.Head[a]
14:    Map.InsertOrUpdate(v, w)
15:    tOffset = blockDim
16:    tOffsetSum += tOffset
17: return {Map.keys(), Map.values()}
```

subtracts u's original degree from $tOffset$. This process is repeated iteratively, until the original degree of $tVert$ is greater than the current $tOffset$, which gives them a unique vertex and adjacency offset within the linked adjacency lists. This allows the thread to retrieve the neighbor and weight of the associated edge for insertion into the hash table. For super-vertices with a Degree larger than the number of processing threads in a warp/block ($blockDim$), threads reset their $tOffset$ to $blockDim$ and continues the process from their current location in the linked adjacency list. When the thread's offset is greater than the total $C.degree[u]$ of super-vertex u, the thread has reached the end of the linked adjacency list and is done with its portion of the expansion.

Algorithm 4 GPU Sorting Parallelization of Adj. Expansion

```
1: Input: CSR Graph G, CM-Graph C, Vertex u
2: {A, W} = ExpandAdjacencies(C, G, u)
3: {A, W} = RadixSort(A, W)
4: {A, W} = ReduceByKey(A, W)
5: return {A, W}
```

We note that the process given in Algorithm 3 can have a significant amount of thread contention for hash table insertions. Hence, we have also created a lock-free and atomic-free variant for GPU adjacency expansion, given in Algorithm 4. As we will discuss in our experimental results, this variant tends to be more performant for large degree vertices and with larger thread groups. This algorithm has three phases. Initially, arrays are filled with the Head[v] values (for A) and associated weights (for W) for all v in the linked adjacency list of super-vertex u. This process is similar to the one given above for our hashing algorithm, though we compute unique offsets in the A, W arrays for each adjacency/weight instead of performing hash table insertions. We then use a radix sort on the vertex IDs in A, with sorting of W simply mirroring the index changes for values in A. Finally, we then perform a reduce-by-key operation, which sums up values in W for each associated 'key' in A and places these values into the final reduced adjacency and weight arrays. These latter two phases utilize atomic-free prefix-sums-based operations as key subroutines. We must omit explicit details about these phases for space considerations, but the broader routines are well-known GPU operations.

G. Analysis

We analyze the complexity of our methods using standard big-O notation. We consider both memory complexity and time complexity for a serial implementation. In addition, we consider parallel work and depth complexity, where 'depth' describes the longest sequence of serial dependencies. Depth can be considered the best-case parallel time given infinite parallel resources, which is generally bounded below as $(n+m)$ parallel processing units for graph computations.

The memory complexity of our coarsening struct is $O(n)$ for the number of vertices of the graph. This is constant, regardless of the number of coarsening levels. When performing explicit adjacency expansion we require additional memory up to the maximum possible degree in the current graph level, which is bounded by the number of heads for that level or simply $O(n)$ in the worst case. In parallel, when there are multiple concurrent adjacency expansions, the worst case memory is required when all edges in G are being considered at once, giving us $O(m)$ complexity, though in practice it is much less than that. Regardless, the memory complexity for all of our methods is the same as the $O(n+m)$ simply required for storage of the original graph.

The time complexity of merging of v into u is dependent on the number of sub-vertices for both, as each Head value requires updating. An upper bound for number of sub-vertices would be 2^k for matching-based coarsening, where k is the level of coarsening and is generally bounded by $\log n$ in real-world settings. However, the time/work complexity for all merges would be $O(n)$, which improves on the $O(n+m)$ required for explicit graph reconstruction. Though, we note that with explicit coarse graph construction, the size of the graph and subsequent complexities decrease with each level.

Parsing the adjacency of a super-vertex also requires traversal of all merged vertices along with an examination of all of their adjacencies. Assume the length of an expanded adjacency is $a = 2^k d_{max}$ in the worst case, where d_{max} is the maximum degree in the graph. An upper bound on work complexity (and parallel depth for naive CPU parallelization) would be $O(a)$ for matching-based coarsening. Over the entire graph, expanding all adjacencies for all Heads requires $O(m)$ work. Our sorting-based GPU algorithm has three primary steps: expanding all adjacencies into their Head values, sorting all Heads and their edge weights, and then doing a reduce-by-key operation. We can expand adjacencies in $O(k^2)$ parallel depth, while radix sort has a parallel depth of $O(\log a \log n)$, as we do a prefix sums across the expanded adjacency list and require $\log_2 n$ radix steps. The final reduce-by-key requires only $\log a$, giving us an overall parallel depth of $O(2^k + \log a \log n)$. For hashing-based parallelization, we have a possible worst-case occurring when only a single unique key (super-vertex)

appears across the full linked adjacency, resulting in a depth of $O(a)$, same as with naive CPU parallelization.

III. Experiments

We perform our CPU testing on the `bella` server at RPI's High Performance Combinatorics and Graph Analytics Laboratory. `bella` has dual AMD EPYC 7742 64-Core Processors with 2 TB DDR4. For GPU experiments, we use a 40 GB NVIDIA A100 GPU on the lab's `zepy` GPU server.

We select the largest 20 matrices (see Figures 2 and 4) from the SuiteSparse Matrix Collection [23] as of July 2024, sorted using nonzeros, for our testing. For our serial tests, we use the 9 largest instances as well as the largest mesh (nlpkkt240). As most of these graphs are too large for our GPU memory, we use the rest of the largest 20 for parallel testing. We consider all edges as undirected and utilize edge weights if available. We otherwise use unit edge weights, and we use unit vertex weights for all graphs.

We initially compare CM-Graph in serial directly against the native METIS implementation. We use the 1-hop METIS functions for both Random (greedy) matching (`Match_RM`) and Sorted Heavy-Edge Matching (`Match_SHEM`) for merge pairs during coarsening. All variables and algorithmic details are the same, except for the differences in how the CM-Graph adjacency structure is accessed and how merges are performed versus how the METIS code builds a new coarsened graph. We compile both codes with 64 bit integer types and `-O3` optimization using `g++ (Ubuntu 9.4.0-1ubuntu1 20.04.2)` for a direct comparison. For our parallel tests, we are able to compile CM-Graph with 32 bit types to fit more test instances into GPU memory. We use `nvcc Cuda compilation tools, release 10.1, V10.1.243` for the GPU codes and the above `g++` with `-fopemp` for the parallel CPU code.

A. Memory Savings

Our first experiment is the total peak memory usage during coarsening. We compare our method against METIS using the two discussed access patterns, with Random Matching and coarsening utilizing the "linked adjacency" method (Algorithm 1) and Sorted Heavy-Edge Matching utilizing the "adjacency construction" method (Algorithm 2).

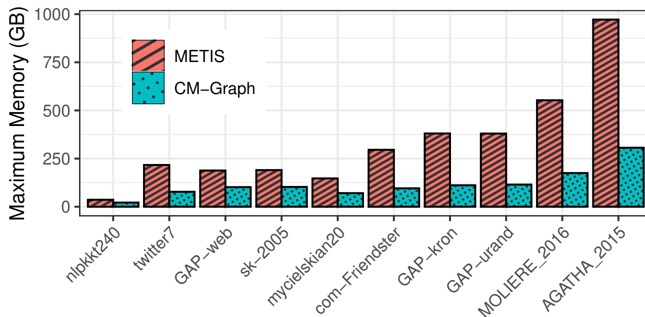

Fig. 2. Maximum memory utilization of METIS and CM-Graph measured when performing Sorted Heavy-Edge Matching and coarsening.

We plot the results for Sorted Heavy-Edge Matching (the results are similar for Random Matching) in Figure 2. We note that our approach saves a significant amount of memory, averaging a somewhat consistent savings of 2.7× across all tests. This savings is primarily due to not having the overhead of storing a graph structure for each coarsening level.

B. Serial Performance

We next consider performance in terms of time to solution. We output the time for coarsening using both Random and Sorted Heavy-Edge Matching, and we plot the results for all datasets in Figure 3.

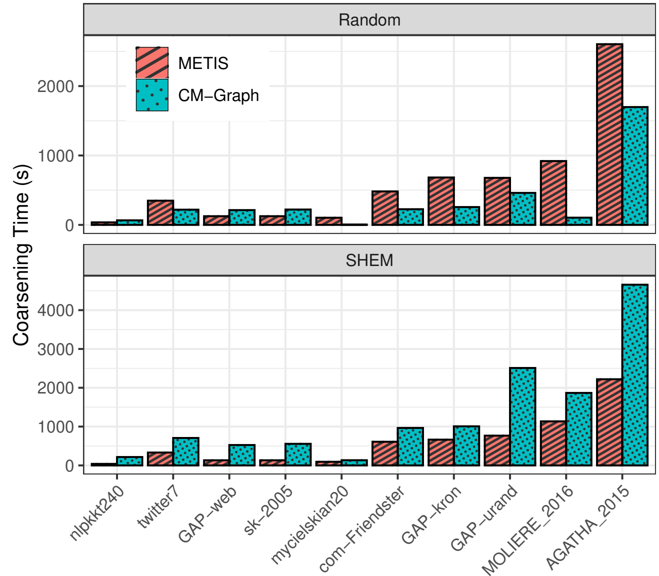

Fig. 3. Coarsening time comparison between METIS and CM-Graph for Random and Sorted Heavy-Edge Matching (SHEM) and coarsening.

We note with surprise that our approach can actually save a significant amount of time in addition to memory overheads when performing direct adjacency accesses during Random Matching and coarsening. We note that this savings is attributable to not having to reconstruct a new graph on each level. For Random Matching-based coarsening, we observe an average speedup of 4.8× (geometric mean of 2.0×). We note the performance is quite variable across instances, from nearly a 2× slowdown on `nlpkkt240` to a 28× speedup on `mycielskian20`, with a higher speedup on more irregular inputs.

For Heavy-edge Matching, we note an average and less variable slowdown of approximately 2.2× with our method. This is attributable to the extra work of having to expand the entire linked adjacency chain and perform the edge weight reductions on each level. In the case of Sorted Heavy-Edge Matching, we need to reduce edge weights over every unique $C.Head[v]$ in the adjacency chain. Whether this tradeoff between time-to-solution and memory overhead is "worth it" will be naturally application-specific. On GPUs or other memory-constrained devices, such a tradeoff might make sense in order to keep all algorithmic data in device memory.

C. Parallel Performance

We now consider our approaches for fine-grained parallelization of adjacency expansion. We compare our serial method on CPU (baseline for speedups), coarse-grained parallelization on CPU (CPU-Coarse), and our two fine-grained parallelizations on GPU (GPU-Hash, GPU-Sort) using parallelized implementations of Heavy-Edge Matching. During testing, we also noticed performance differences for GPU-Hash and GPU-Sort based on vertex degree, with the sorting method being more performant for large degree vertices and hashing being faster for low degree vertices. Hence, we also implemented a basic heuristic (GPU-Combined), where we assign vertices with degrees higher than the warp size of 32 for sorting and vertices with lower degrees for hashing.

For fine-grained parallelization on GPU, we assign a warp of 32 threads to each vertex for adjacency expansion. We run 256 threads on CPU, which is the maximum supported by our hardware. We compare speedups against serial times for adjacency expansion. The core Sorted Heavy-Edge Matching and coarsening algorithm for all tests is otherwise the same. Optimizing the coarsening algorithm itself is not the focus of this current effort, though incorporating our methods within an optimized GPU framework [2] will make for interesting future work.

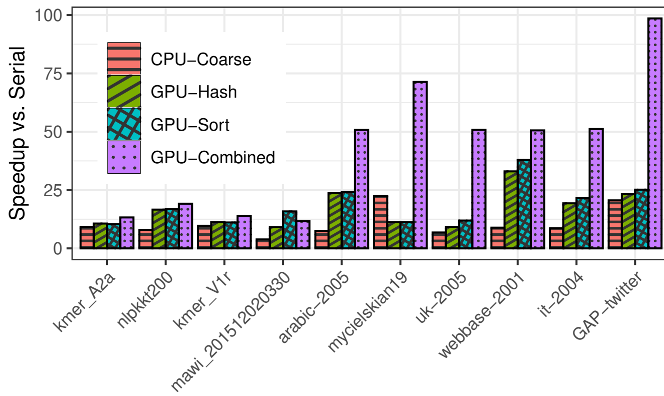

Fig. 4. Parallel speedup for coarsening with CM-Graph comparing CPU coarse parallelization and GPU fine parallelization of our adjacency expansion algorithms within Sorted Heavy Edge Matching. Speedups are relative to serial CPU adjacency expansion.

Given in Figure 4 is the speedup of our parallelization approaches relative to serial. Of particular note is how our GPU-Combined heuristic is considerably more performant on the larger inputs (size of input increases from left to right), with an overall average speedup versus serial of $43\times$, while most other methods have comparably modest speedups. Though not shown for space, we also note that maximum memory utilization is relatively low, being only marginally higher that what is needed for expanding the maximum degree super-vertex, even with a high thread concurrency. These observation are attributable to the fact that most of these inputs have particularly skewed degree distributions with a lot of low degree vertices and a few very high degree vertices. The input `mawi_201512020330` also has this characteristic, though it has vertices with degrees of the order of the total number of vertices, which dominates its processing time. This overall suggests that our heuristic and overall methods are quite effective, and nontrivial parallelization strategies are necessary to get optimal performance for coarsening these benchmarks in general. Additional analysis and optimization of our heuristic combined with variable thread counts based on vertex degrees is a very good direction for future work.

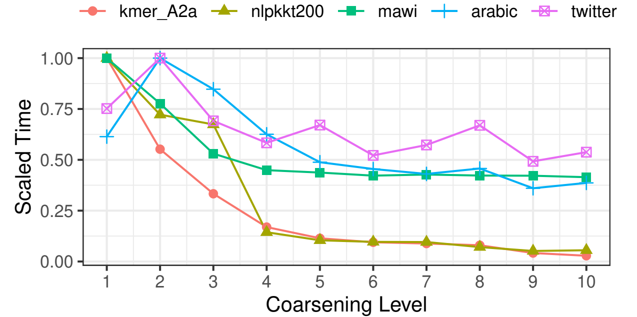

Fig. 5. Scaled coarsening time versus coarsening level for Sorted Heavy Edge Matching on representative test instances. The plotted time values are relative to the slowest iteration for each instance.

Our final experiments measure time per coarsening level, plotted in Figure 5 for representative instances. We scale the time per level proportionally to the maximum per-level time for each input after running heavy-edge matching and coarsening for a fixed 10 levels. We note that our method accelerates as the graph coarsens, with the 10th level running $20\times$ faster than the 1st level for the more regular inputs. This indicates that it will not be prohibitive to directly run a graph partitioning or other algorithm directly on a coarsened graph using CM-Graph in some instances, instead of instantiating a new coarsened representation. An obvious next step is direct integration of partitioning algorithms after coarsening.

IV. Conclusions

We implemented CM-Graph as a low-overhead data structure for shared-memory graph coarsening. CM-Graph empirically offers a several-fold savings in memory and is relatively lightweight, offering either a $2\times$ speedup or $2\times$ slowdown in real-world applications, depending on how the adjacency structure needs to be accessed. We also introduce a parallelization of our method for GPU, which similarly offers competitive performance. Future work is considering extending this work to distributed memory, further optimizing our GPU algorithms, and utilizing this method for graph partitioning.

V. Acknowledgments

This work is supported by the National Science Foundation under Grant No. 2047821 and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute under Contract No. DE-SC0021285 at the Rensselaer Polytechnic Institute, Troy NY.

REFERENCES

[1] M. Kumar, A. Sharma, and S. Kumar, "A unified framework for optimization-based graph coarsening," *Journal of Machine Learning Research*, vol. 24, no. 118, pp. 1–50, 2023.

[2] M. S. Gilbert, S. Acer, E. G. Boman, K. Madduri, and S. Rajamanickam, "Performance-portable graph coarsening for efficient multilevel graph analysis," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 213–222.

[3] I. Safro, P. Sanders, and C. Schulz, "Advanced coarsening schemes for graph partitioning," *Journal of Experimental Algorithmics (JEA)*, vol. 19, pp. 1–24, 2015.

[4] I. Dhillon, Y. Guan, and B. Kulis, "A fast kernel-based multilevel algorithm for graph clustering," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 629–634.

[5] Z. Huang, S. Zhang, C. Xi, T. Liu, and M. Zhou, "Scaling up graph neural networks via graph coarsening," in *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, 2021, pp. 675–684.

[6] M. Fahrbach, G. Goranci, R. Peng, S. Sachdeva, and C. Wang, "Faster graph embeddings via coarsening," in *international conference on machine learning*. PMLR, 2020, pp. 2953–2963.

[7] O. E. Livne and A. Brandt, "Lean algebraic multigrid (lamg): Fast graph laplacian linear solver," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. B499–B522, 2012.

[8] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*. Elsevier, 2000.

[9] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid methods*. SIAM, 1987, pp. 73–130.

[10] J. Chen, Y. Saad, and Z. Zhang, "Graph coarsening: from scientific computing to machine learning," *SeMA Journal*, vol. 79, no. 1, pp. 187–223, 2022.

[11] B. F. Auer and R. H. Bisseling, "Graph coarsening and clustering on the gpu." *Graph Partitioning and Graph Clustering*, vol. 588, no. 223, p. 2, 2012.

[12] C. Cai, D. Wang, and Y. Wang, "Graph coarsening with neural networks," *arXiv preprint arXiv:2102.01350*, 2021.

[13] Y. Jin, A. Loukas, and J. JaJa, "Graph coarsening with preserved spectral properties," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2020, pp. 4452–4462.

[14] C. Brissette, A. Huang, and G. Slota, "Parallel coarsening of graph data with spectral guarantees," *arXiv preprint arXiv:2204.11757*, 2022.

[15] ——, "Spectrum consistent coarsening approximates edge weights," *SIAM Journal on Matrix Analysis and Applications*, vol. 44, no. 3, pp. 1032–1046, 2023.

[16] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, vol. 7933. Springer, 2013, pp. 164–175.

[17] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[18] G. M. Slota, S. Rajamanickam, and K. Madduri, "A case study of complex graph analysis in distributed memory: Implementation and optimization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 293–302.

[19] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, 2016, pp. 1–12.

[20] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–35, 2018.

[21] G. M. Slota, S. Rajamanickam, and K. Madduri, "High-performance graph analytics on manycore processors," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 17–27.

[22] G. M. Slota, J. W. Berry, S. D. Hammond, S. L. Olivier, C. A. Phillips, and S. Rajamanickam, "Scalable generation of graphs for benchmarking hpc community-detection algorithms," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[23] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.