DMM-GAPBS: Adapting the GAP Benchmark Suite to a Distributed Memory Model

Zach Hansen
University of Nebraska Omaha
zachhansen@unomaha.edu

Brody Williams
Texas Tech University
brody.williams@ttu.edu

John D. Leidel
Tactical Computing Laboratories
jleidel@tactcomptlabs.com

Xi Wang
RIOS Laboratory
xi.w@rioslab.org

Yong Chen
Texas Tech University
yong.chen@ttu.edu

2021 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-2369-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPEC49654.2021.9622817

***Abstract*—Due to the ability of graphs to model diverse real-world scenarios such as social networks, roads, or biological networks, effective graph processing techniques are of critical importance to a wide array of fields. As a consequence of the growth of data volumes, some graphs have already outgrown the memory capacities of single servers. In such cases, it is desirable to partition and keep the entire graph in a distributed memory space into order to bring the resources of a computing cluster to bear on the problem. This approach introduces a number of challenges, such as communication bottlenecks and low hardware utilization. However, it is difficult to effectively measure the impact of innovations addressing these challenges due to a lack of standardization in the domain of distributed graph processing. This research study was inspired by, and builds off of, the widely-used GAP Benchmark Suite (GAPBS), which was developed to provide an effective baseline and consistent set of evaluation methodologies for shared memory multiprocessor graph processing systems. We design and develop a new benchmark suite called DMM-GAPBS, a distributed-memory-model GAPBS. We adapt the GAPBS graph building infrastructure and algorithms, but utilize OpenSHMEM to enable a distributed memory environment, in the hope of providing a modular, extensible baseline for the distributed graph processing community. In order to showcase our design and implementation for processing graphs that cannot fit within a single server, we present the results of executing the DMM-GAPBS benchmark kernels on two large synthetic graphs distributed across sixteen nodes of an enterprise class system.**

***Index Terms*—Benchmark, GAPBS, Graph, Distributed Memory, Performance**

I. Introduction

Graphs, and similar data structures derived from them, have formed a vital component of software infrastructures for the greater part of the modern computing era. Based on the mathematical concept of the same name, graphs provide an easily comprehensible abstraction for modeling relationships. Today, graphs are ubiquitous within fields such as data mining, information retrieval, and scientific computing, where they are used to solve complex problems. While many useful problems can be modeled with relatively small graphs, modern computational workloads often require the construction and processing of graphs whose memory footprint exceeds the resources provided by a single node. The advent of the Big Data era has further exacerbated this trend, while also expanding interest in graph processing beyond traditional venues such as academia and other scientific institutions.

Unfortunately, challenges associated with graph processing are further compounded when graphs are coupled to distributed memory and parallel processing models. The challenges include poor data locality and frequent irregular data access [1]. In a distributed setting, it is desirable to distribute graph vertices and/or edges across distributed resources such that the principles of data locality can be effectively leveraged. However, a practical, generalizable methodology for doing so remains elusive. Similarly, execution of graph algorithms in distributed, parallel environments necessitates the use of some form of synchronization. These synchronization operations, which may constitute non-trivial portions of executed instructions [2], are widely understood to be an impediment to performance, regardless of implementation [3]. As such, it is desirable, but often difficult, to minimize the use of these operations. Moreover, the random memory access patterns that graphs most often exhibit make efficient processing of large-scale graphs difficult.

As we prepare to enter the post Moore's Law era, system architects have been forced to embrace an increasingly heterogeneous approach when designing future platforms. Given the diverse nature of this new paradigm, the importance of hardware/software co-design is paramount. Moreover, an effective means for measuring the performance of this emerging class of architectures with respect to graph workloads is crucial. Motivated by the above observations, in this work we introduce DMM-GAPBS, a graph benchmark suite for distributed memory systems. DMM-GAPBS is a graph benchmark suite that incorporates several kernels which exemplify the behavior of common graph-related workloads. Adapted from the GAP Benchmark Suite (GAPBS), we believe that DMM-GAPBS represents an important step towards a portable, standardized, benchmark for graphs in distributed memory environments.

The primary contributions of this work are an open source prototype of DMM-GAPBS[1], a detailed description of its implementation and the design choices made, and a brief experimental evaluation.

II. Background

Related Work and Motivation. Shao et al. argue that, due to irregular access patterns, distributed processing that

[1]Available at https://github.com/tactcomplabs/DMM-GAPBS.git

keeps the entire graph in memory is the best approach for processing large graphs [4]. Similarly, Buluc and Gilbert conclude that out-of-core approaches are "infeasible" after comparing experimental results from [5]–[7]. This requires extending existing parallel algorithms from shared memory models of computation such as the Parallel Random Access Machine (PRAM) [8] into distributed memory models such as a Partitioned Global Address Space (PGAS). However, this approach is not without limitations. Many graph algorithms experience communication bottlenecks and consequently make poor use of the available hardware [9]. In a distributed setting, these communication overheads often become an impediment.

Orthogonally, the versatility of graphs and the difficulties associated with distributed graph processing has given rise to several graph processing frameworks that vary greatly in design. Several notable efforts include PGBL [10], GAPDT [11], Pregel [12], and Combinatorial Blas [5]. Parallel GBL is a flexible, generic library written in C++ that provides distributed, parallelized graph processing data structures and algorithms. When this framework uses distributed adjacency lists for representing the graph, communication occurs via MPI using the Bulk Synchronous Parallel (BSP) model [13].

The Graph Algorithm and Pattern Discovery Toolbox (known as KDT) provides MATLAB-based interactive processing of large graphs (represented as sparse matrices). Pregel is a distributed programming framework with an emphasis on fault-tolerance and abstraction of distribution details. It provides an API for programming graph algorithms with an underlying message passing model. Finally, Combinatorial BLAS exploits the relationship between graphs and sparse matrices. Since many graph operations can be represented as linear algebraic operations on sparse matrices, the authors provide a set of linear algebra primitives to facilitate the rapid implementation of scalable graph algorithms.

Our research study designs and develops a new benchmark suite, inspired by the GAPBS graph building infrastructure and algorithms, but uses OpenSHMEM for a distributed memory environment. The objective of this research is to provide a modular, extensible baseline for the distributed graph processing community.

Our philosophy differs fundamentally from the previously described frameworks in three core ways. First, whereas the others are designed to abstract away the details of distribution and synchronization for the benefit of graph application developers, we attempt to make these details as visible and intelligible as possible. It is our hope that this visibility and transparency will inspire others to modify selected pieces of our infrastructure to test theoretical innovations. Second, whereas KDT and Combinatorial BLAS represent graphs with sparse matrices and PGBL and Pregel use message passing models for communication, we utilize distributed adjacency lists accessible in the OpenSHMEM symmetric heap for asynchronous operations. Finally, we extend the well-known GAPBS benchmarking methodology, graphs, and kernels to a distributed memory setting.

GAP Benchmark Suite The GAP Benchmark Suite was introduced to address the critical need for a set of standardized graph application benchmarks [14], [15]. Designed to emulate the behavior of several graph algorithms commonly utilized across application domains, the GAPBS has proven to be an invaluable tool for the community. Naturally, GAPBS has been extensively utilized for studies related to graphs and graph processing [16], [17]. More surprising is the fact that GAPBS has also been utilized in dissimilar fields such as security [18], [19]. Moreover, GAPBS has already been broadly leveraged in the design of future architectures as discussed in the previous section [20]–[22].

The GAPBS itself is composed of a set of evaluation methodologies, input graphs, and graph kernel specifications with corresponding implementations written in C++ and parallelized with OpenMP. The Breadth-First Search (BFS) kernel implements direction-optimizing BFS and tracks parents for reachable vertices [23]. The Betweenness Centrality (BC) kernel approximates betweenness centrality scores for every vertex. That is, it computes the fraction of shortest paths that pass through a given vertex by finding shortest paths originating from a subset of vertices. The core algorithm is the Brandes algorithm [24] with enhancements by Madduri et al. [25]. Connected Components (CC) uses the Afforest subgraph sampling algorithm to produce a component label for each vertex [26] [27]. Two vertices should share a component label if and only if they belong to the same (weakly) connected component (an undirected path exists between the two). Let $N^+(v)$ denote the outgoing neighborhood of vertex v (vertices with an incoming edge from v) and let $N^-(v)$ denote the incoming neighborhood of v. The PageRank (PR) score for a vertex v and a damping factor d is defined as

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N^-(v)} \frac{PR(u)}{|N^+(u)|}$$

These scores are calculated using a common, (not state-of-the-art) iterative approach that is similar to sparse-matrix vector multiplication. The Triangle Counting (TC) kernel counts the number of cliques of size three present in an undirected graph by finding overlap between sorted neighbor lists [28]. As an additional optimization, graphs where very few nodes have very large degrees are relabeled according to their degree. Finally, the Single Source Shortest Paths (SSSP) kernel implements the δ-stepping algorithm [29] with a bucket fusion optimization from Zhang et al. [30] to produce distances from the source for each vertex.

One limitation of the GAPBS is its focus on relatively small graphs. In its current form, GAPBS is capable of execution using only graphs that will fit into a single physical memory space. Although already immensely useful, this prevents the GAPBS from being applied to the larger problems that drive the design of future systems. As detailed in the next section, we build DMM-GAPBS as an adaptation of the GAPBS, using OpenSHMEM [31], [32] to expand to its applicability to distributed memory systems and larger graphs.

III. Design & Implementation

In this section, we detail the important design changes we made to adapt the GAPBS reference code to a PGAS model. Our goal is to enable the processing of larger graphs than the original implementation could store, and, as such, we

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

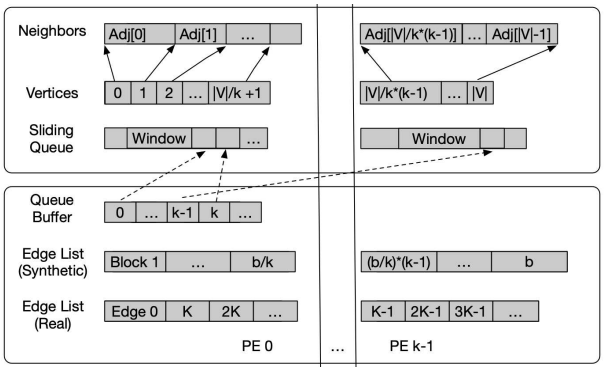

Fig. 1: Partitioned data structures in the symmetric heap (Neighbors, Vertices, and Sliding Queue) and the local heap (Queue Buffers and Edge Lists). PE 0 is flushing its Queue Buffer contents to the incoming region of the Sliding Queue following the visible window.

particularly emphasize how we divided memory requirements across processing elements.

Let p be the ordinal of the processing element (PE) executing the instruction, and let k be the number of processing elements involved in the computation. In our implementation, all PEs are part of the active set [32]. Let $G = (V, E)$, where V is the sequence of natural numbers $0, \ldots, |V|-1$. Let $Adj[i]$ be the adjacency list of vertex i.

A. Data Structures

a) Partitions: Partitioning of the graph occurs in two ways: naively and round robin. Vertices are assigned naively to PEs such that vertices in the range $\left[\left\lfloor\frac{|V|}{k}\right\rfloor * p, \left\lfloor\frac{|V|}{k}\right\rfloor * (p+1)\right)$ are assigned to PE p for all $p \in \{0, \ldots, k-2\}$ and vertices in the range $\left[\left\lfloor\frac{|V|}{k}\right\rfloor * (k-1), |V|\right)$ are assigned to PE $k-1$. This is very similar to the partitioning approach taken by PGBL for representing distributed adjacency lists. The "partition width" of a naive partitioning of n elements (denoted w_p^n) is $\left\lfloor\frac{n}{k}\right\rfloor$. The "max width" of a naive partition (denoted w_m^n) is $n - (k-1) * w_p^n$. Equivalently, it is the number of elements assigned to PE $k-1$. Unless otherwise specified, when we describe work as being "naively divided between PEs" it means that PEs $0, \ldots, k-2$ process w_p^n elements and PE $k-1$ processes w_m^n elements. Figure 1 provides an overview of this important data structure partitioning scheme.

b) P-Vectors: The original implementation included a custom vector class that provided the option to initialize the internal array in parallel. We extended this class with an option to allocate the internal array in the symmetric heap. Note that typical usage has P-Vectors pushing back different elements with no attempt to ensure the arrays have the same contents across PEs.

c) Bitmaps: The custom Bitmap class was also extended to support symmetric memory versions. Since Bitmaps are such a compact representation, unlike the other data structures, we do not partition the symmetric versions in our current implementation. PEs can only get or set the values of bits within their own copy of the bitmap. Bitmaps can be merged when necessary to provide every PE with an up-to-date record of all bits that were set in the preceding phase. For example, during direction-optimizing breadth-first search, in the bottom-up step children search the active frontier (represented as a bitmap) for parents. Each PE scans the adjacency list of all unvisited vertices assigned to it, and sets a vertices' bit if it has a visited parent. Thus, each PE only sets bits of vertices assigned to it during a bottom-up step and the results can be safely combined at the end with an OR-TO-ALL collective call.

d) Sliding Queues: Kernels such as Breadth-First Search and Betweenness Centrality utilize a Sliding Queue data structure for representing frontiers, which, in the original implementation, is a double-buffered queue to which parallel threads append queue buffer contents in bulk. In our implementation, the role of threads is replaced by OpenSHMEM PEs. The sliding queue itself resides in the symmetric heap, and is partitioned similarly to P-Vectors (each PE allocates room for w_m^n elements to partition a queue with maximum capacity n). Access to the queue is controlled by a mutual exclusion lock. PEs fill up local-memory queue buffers, then acquire the lock[2] and distribute the contents of their buffer round robin to all PEs. If the queue on the requested destination PE is full, the calling PE searches linearly through ascending PEs until one with space is found. Since the round robin distribution always begins with PE 0, each of n elements will always find space in one of the queues. This performs load balancing[3] at the cost of high communication overhead.

e) Tournament Trees: In order to support sorting in a distributed environment, we added a tournament tree data structure. Each PE contributes a leaf to a complete binary tree, which is built in symmetric memory. One PE is designated as the "leader", and pops the root repeatedly, fetching data from the PE that supplied the root to rebuild the tree. When a new PE needs to lead (to fill its portion of the sorted list), the current leader transfers the complete contents of the tree to the new PE.

B. Graph Building

a) Reader: While reading in edge lists from a file, every PE reads the entire file but selects only a subset of edges to store in its local edge list in a round robin fashion. PE 0 takes the first edge encountered edge, PE 1 takes the second edge, and so on. This reduces the space required for the edge list during graph construction by a factor of k. Currently only files with the extension .el (edge list), .wel (weighted edge list) and .gr (graph) are supported.

b) Generator: The generator supports generation of two types of random graphs: the Uniform Random [33] and Kronecker graphs [34] using Graph 500 parameters [35]. For both approaches, the RNG is seeded every B edges with a fixed seed that is incremented by the block number, which divides the generation of edges into b "blocks." In order to remain deterministic with the original implementation, we naively

[2]A lock-free version that uses atomic fetch and add instead is currently under development.

[3]As long as k is less than the queue buffer size, which by default is 16,384.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

partitioned these b blocks between k PEs. Consequently, when $|E| < B$, or if $b < k$, then PE $k-1$ assumes all responsibility for generating the edge list. For anything but trivially small graphs this approach is approximately balanced, but the balancing can always be improved by reducing B. The Kronecker generation requires permuting the complete edge list. Currently, the only way to maintain determinism with the original implementation during this permutation involves an array of length $|V|$. We included an alternative approach which reduces this required space to $w_m^{|V|}$, but is not deterministic (which inhibits verification). Including the verification flag when running a kernel automatically makes the graph building deterministic.

Single Source Shortest Path requires weighted graphs, but the synthetic graphs and some input graphs are unweighted. The Generator therefore supports the insertion of deterministic weights. If the edge list was read in from a file, then the generation of weights is distributed in the same manner as before: b blocks are divided naively among k PEs, with the remainder being assigned to PE $k-1$. For normal quantities of $|E|, B, k$, this requires each PE allocating an array of size $\lfloor \frac{b}{k} \rfloor * B$ on the symmetric heap. If the edge list was generated, weights are directly inserted into the edge list with no extra allocations required. In both cases, our implementation is deterministic with the original.

c) Builder: Once an edge list is read or generated, the Builder constructs the graph. The original implementation stored all the adjacency lists in a single array (size $|E|$), with an array of pointers to locations in that array to distinguish between the outgoing neighborhoods of different vertices (size $|V|$). We build the graphs in symmetric memory to enable PEs to access the neighborhoods of vertices not assigned to them. However, this requires all PEs to allocate the same amount of space for their adjacency list. Thus, if s_p is the first vertex assigned to PE p and e_p is the final vertex assigned to p, then the amount of space on each PE required for the outgoing neighbor CSR format is

$$\max_{p \in \{0,\ldots,k-1\}} \sum_{i=s_p}^{i \leq e_p} |Adj[i]|$$

The incoming neighborhoods are stored in the same manner. Duplicate edges and self-loops are removed (each PE processes its assigned vertices) and the neighborhoods are sorted in non-descending order.

C. Kernels

In the following section we detail important changes to the kernels. Explaining each algorithm in detail is beyond the scope of this paper, so we refer the reader to the source code of the GAPBS and DMM-GAPBS implementations, as well as to the papers in which these algorithms were introduced. Our main guiding principle while adapting these algorithms was faithfulness to the behavior of the original GAPBS, in this sense what follows is a "direct conversion" from a shared memory implementation to a distributed memory implementation. Alternative algorithms specifically designed for distributed memory models certainly exist, and comparing their performance with our implementation will be an interesting direction for future work.

a) Breadth-First Search (BFS): We adapted direction-optimizing BFS to a distributed memory setting by parallelizing the top-down (TD) and bottom-up (BU) steps. During TD, each PE processes their local portion of the frontier (a Sliding Queue), exploring outgoing neighborhoods and adding discovered children to their local queue buffers. These buffers are flushed when full and when the step ends, preparing the Sliding Queue for the next iteration. During BU, each PE searches for parents for all nodes assigned to that PE by the naive vertex partitioning. When the frontier is converted from a Sliding Queue to a Bitmap during the switch from TD to BU, each PE sets the corresponding bit for each node in that PE's portion of the queue. Then the symmetric bitmaps are merged. When converting from a Bitmap to a Sliding Queue, each PE searches for set bits occurring within its vertex partition range and pushes the associated node to the local queue buffer. The frontier and parent array are both reduced from size $|V|$ to size $w_m^{|V|}$ (on each PE).

b) Connected Components (CC): The algorithm begins by approximating the components, then samples the resulting component labels to determine the largest intermediate component. Distributing the selection of nodes whose labels are sampled approximately evenly across PEs would reduce the randomness, and the original implementation only sampled 1024 elements. As such, we did not parallelize this step and had PE 0 handle all the work. The Link(u,v) subroutine was not parallelized either as its only purpose is to place two nodes (u and v) in the same component, however each PE only calls Link on a u within its own partition. When a PE calls the Compress subroutine, it compresses the component labels of the range of vertices assigned to that PE. The component label array is partitioned into P-Vectors of size $w_m^{|V|}$ on each PE. The component label array (size $|V|$) is partitioned into arrays of size $w_m^{|V|}$.

c) Triangle Counting (TC): The Triangle Counting algorithm uses a heuristic to estimate if graphs are scale-free with a sufficiently high average degree to warrant re-building the graph with vertices relabeled by their degree. In order to sort the vertices into non-ascending order by degree, each PE sorts the nodes within their partition, then the partially sorted lists are combined using k-way merge and a tournament tree. This requires (at one time) three auxiliary arrays of size $w_m^{|V|}$. During the actual triangle counting phase (OrderedCount), all PEs attempt to find ordered triangles originating from their assigned vertices.

d) Page Rank (PR): Unlike the other kernels, the original implementation of the Page Rank kernel uses a very common, naive algorithm instead of a state-of-the-art approach[4]. Our adaptation is similarly simple: work is naively divided between PEs during each iteration.

e) Single-Source Shortest Paths (SSSP): The Delta Stepping algorithm functions similarly to Dijkstra's algorithm, but it maintains an array of buckets of width Δ instead of a

[4]The GAPBS was very recently updated to include an alternative PR implementation. Note that we adapted the legacy version (the sparse-matrix vector multiply version).

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

priority queue of unsettled nodes. The frontier in this kernel is represented by a P-Vector, originally of size $|E|$, but reduced in our implementation to $w_m^{|E|}$. Each iteration of the main loop (which iterates until all buckets are empty), PEs $0, \ldots, k-2$ process w_p^n of the n active nodes added to the frontier in the prior iteration, while PE $k-1$ processes w_m^n nodes. Edges emanating from nodes in the frontier are relaxed, and any encountered vertices whose distances can be improved are added to PE-local buckets. At the end of this phase of the iteration, all PEs vote to determine the smallest non-empty bucket. In the next phase, all threads (in the original implementation) or PEs (in our implementation) copy their local bucket contents into the shared frontier to prepare for the next iteration. Since our frontier is partitioned, this introduces some additional complexity. Let n be the number of nodes in the current bucket summed across all PEs. Now the PEs naively partition and distribute these n elements across the frontier (w_p^n elements in the frontier on PEs $0, \ldots, k-2$, w_m^n elements on PE $k-1$). This approximately balances the work for the next iteration at the cost of high communication overhead.

f) Betweenness Centrality (BC): The P-Vectors in BC are reduced from $|V|$ to $w_m^{|V|}$, as is the Sliding Queue representing the BFS frontier. The Brandes algorithm performs a number of parallel breadth-first searches from varying sources to estimate the number of paths through each node. During a given BFS, at each depth, each PE calculates path counts for nodes in its portion of the partitioned frontier. During the main algorithm, each PE converts these path counts into scores for the vertices (nodes) assigned to it. Work for normalizing the scores is divided naively in the same way: PEs normalize scores for nodes in their vertex partition.

D. Verification

Our current verification process across kernels is rudimentary. Since we kept the generation of graphs and weights deterministic, we can rely on the verifiers from the original implementation. Including the verification flag in our implementation prints the result to a file. With the exception of BC and PR, calling a slightly modified version of the original implementation with the same graph parameters builds the same graph in shared memory, but runs the verifier on the saved output instead of executing the original implementation's kernel. In order to avoid precision issues in BC and PR, we compare each of the scores produced by our implementation to the range of values of the corresponding original kernel score plus/minus 0.00001. (This comparison is in keeping with the recommendations of the GAPBS specification with regards to numerical noise [14]). Since the verifier does not run on a distributed graph, it allows us to confirm that the graph building infrastructure as well as the kernels are working properly.

IV. Evaluation

In comparison to shared memory multiprocessor systems, distributed graph processing is notoriously inefficient - the main use-cases are graphs that are too large to fit within the memory of a single node [9]. For this reason, we focus our evaluation on large synthetic graphs whose edge lists require memory footprints that often exceed the capacity of a single node within a cluster. In the following, vertex IDs are represented by 32-bit unsigned integers. The number of vertices is $|V| = 2^s$, where s is the *scale* of the graph. The number of edges is $|E| = |V| * d$, where d is the *degree* of the graph. In our evaluation, we fix the degree to 64 and execute graphs of scale 28 and 29. An unweighted edge (2 vertex IDs) is 8 bytes, consequently these graphs have unweighted edge lists of size (approximately) 137 and 274 GB. SSSP requires weighted edges (12 bytes per edge), thus the corresponding edge list sizes are 206 and 412 GB.

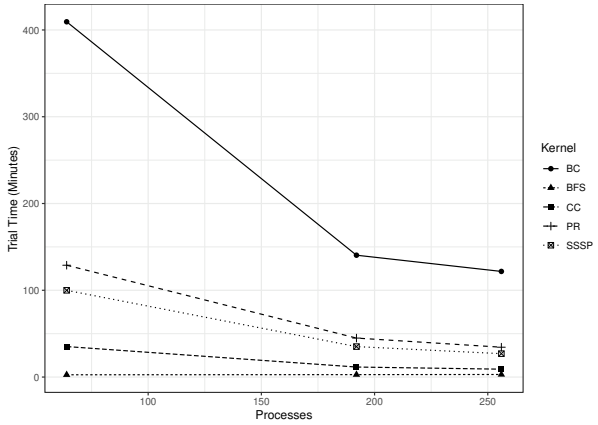

Fig. 2: Timing results for Uniform Random graph of scale 28.

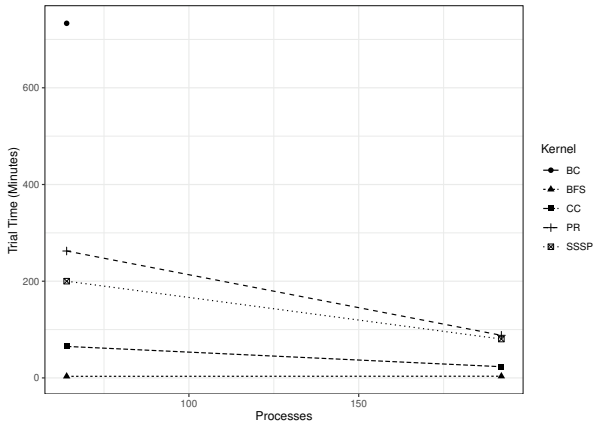

Fig. 3: Timing results for Uniform Random graph of scale 29.

In keeping with our motivation of benchmarking emerging distributed memory architectures, we perform our evaluation of DMM-GAPBS on a Cray Advanced RISC Machine (ARM) platform. Details of the configuration of this system are shown in Table I. Sixteen nodes were used in each trial. The execution times of each kernel for varying Processes Per Node are recorded in Table II. Note that while the kernels all were verified to be functional as detailed in Section III-D, all the execution times of the TC kernel, as well as one BC trial (denoted by -), exceeding the scheduler limits on our test platform. We used small numbers for Processes Per Node (PPN) to ensure that the cumulative size of the non-partitioned data structures (the Bitmaps) did not exceed the available

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

memory of a single node. One difficulty we encountered was determining the amount of symmetric heap space required prior to executing a given kernel. For Uniform Random graphs, we could approximate the heap space required for each PE to represent their portion of the distributed adjacency list with reasonable accuracy. Appropriate parameters for Kronecker graphs were more difficult to discern. As such, our evaluation focuses on trials using Uniform Random graphs.

TABLE I: Evaluation Platform: Per node configuration

Parameter	Configuration
ISA	ARMv8.1
CPU	2x Cavium ThunderX2 9975, 28 cores
Memory	256GiB DDR4
Network Interconnect	Cray Aries, Dragonfly Topology
Operating System	CLE SLES 15.1
Compiler	GCC 10.2.0
OpenSHMEM	OSHMEM 4.1.1 + UCX 1.10.1

Our evaluation made two things evident. First, the difficulty of setting an appropriate heap size makes vertex partitioning schemes that distribute the adjacency list evenly more appealing. While other partitioning schemes may be simpler (such as our naive baseline) or exploit data locality better (such as minimum-cut or community based approaches), variability in adjacency list size across PEs is very damaging to our PGAS-based approach. The wasted memory on PEs that must overallocate symmetric heap space to match the PE with the largest edge list adds up quickly, limiting the size of the graphs that can be processed. Uneven partitioning also limits the usability of the suite by forcing users to tune the symmetric heap size through trial and error. Second, the low hardware utilization (PPN) was primarily necessary to accommodate the Bitmaps. However, Figure 3 demonstrates that most kernels benefit substantially from increased PPN. CC, for instance, experienced a $3.04\times$ speedup when PPN tripled (4 to 12) and a $1.27\times$ speedup when PPN was increased by $1.33\times$ (12 to 16). BC, PR, and SSSP also demonstrated approximately linear speedups as PPN increased for graphs of scale 28 and 29[5]. BFS is the only kernel which demonstrated negligible improvement in execution time as PPN increased. It is possible that increased communication requirements mostly nullified the benefits of additional parallelism for this particular kernel. On the whole, however, the suite scales well with PPN. This further underscores the need to partition the Bitmaps to obtain optimal execution times and to make more efficient use of the available hardware.

Graph	PPN	BC	BFS	CC	PR	SSSP
U-28	16	7303.5	177.0	543.8	2063.5	1616.9
	12	8425.4	167.3	692.4	2694.8	2105.5
	4	24566.3	156.9	2107.9	7732.5	5997.9
U-29	12	–	210.4	1390.2	5284.2	4822.2
	4	44002.9	198.1	3907.7	15747.3	12004.2

TABLE II: Timing results for processing Uniform Random (U) graphs of scale 28-29, degree 64, for each kernel. **PPN** is processes per node. Timing results are in seconds.

[5]An exception is SSSP on U-29, which only experienced a speedup of $2.5\times$ when PPN tripled.

V. Conclusion & Future Work

Our prototype implementation of the DMM-GAPBS presented in this paper provides ample space for further optimizations and improvements. We anticipate a number of enhancements will be made in the future, enhancements which will hopefully be guided by community utilization and feedback. For the time being, we have identified several ways in which the suite can be further improved, some of which are described here. 1) Rebuilding the graph during the execution of the Triangle Counting kernel is a communication-intensive process that requires distributed sorting and re-distribution of node assignments across PEs. A new heuristic needs to be tuned to determine when re-building the graph will actually improve performance in a distributed memory setting. In a similar vein, the direction-optimizing BFS uses heuristically chosen parameters to determine when to switch between TD and BU directions. We use the default parameters in our evaluation, but a more comprehensive search for optimal parameters would likely accelerate this kernel. 2) When we converted the graph building infrastructure, the Builder would construct a graph from the edge list, then allocate space for a compressed version of the graph and rebuild it (removing redundant edges and self-loops). A more recent modification to the GAPBS reference code enables the construction of graphs in-place. Adapting this approach to a distributed setting will be more complicated, but the space savings should be worthwhile. 3) Currently verification is limited to graphs that the GAPBS can also process. Writing reliable distributed memory verifiers will enable us to confirm the successful execution of the kernels on larger graphs. 4) Since one of the primary goals of the DMM-GAPBS is to divide the memory requirements across computing resources, partitioning the vertices into ranges such that the complete adjacency list is evenly distributed will likely be worth the additional pre-processing required to calculate neighborhood sizes. Consistent with one of the primary motivations for this work, we also intend to leverage DMM-GAPBS in our ongoing work on the novel xBGAS microarchitecture extension to the RISC-V instruction set architecture [36].

The breadth and complexity of the topic of distributed graph processing makes consistent baselines and evaluation methodologies essential. With DMM-GAPBS, we have taken preliminary steps towards extending the standardizing effect of the GAP Benchmark Suite to a distributed memory setting. We hope that the transparency of our implementation will enable members of the graph processing community to easily adapt, extend, and modify the kernels and graph building infrastructure. The empirical results in the Evaluation section demonstrate the ability of our implementation to store and process graphs whose memory footprints exceed the capacity of a typical commodity server. They also suggest that the speed of most kernels scales well with increased processes per node, although a more comprehensive evaluation with varied numbers of nodes is in order.

VI. Acknowledgements

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the

National Science Foundation under grants CCF-1409946 and CNS-1817094. The authors would also like to thank Los Alamos National Laboratory for use of the Capulin system. This work is authorized for release under LA-UR-21-28427.

REFERENCES

[1] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007. [Online]. Available: https://doi.org/10.1142/S0129626407002843

[2] X. Wang, B. Williams, J. D. Leidel, A. Ehret, M. Kinsy, and Y. Chen, "Remote atomic extension (rae) for scalable high performance computing," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[3] B. Williams, J. Leidel, X. Wang, D. Donofrio, and Y. Chen, "Circustent: A benchmark suite for atomic memory operations," in *The International Symposium on Memory Systems*, ser. MEMSYS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 144–157. [Online]. Available: https://doi.org/10.1145/3422575.3422789

[4] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of SIGMOD 2013*, Jun 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/trinity-a-distributed-graph-engine-on-a-memory-cloud/

[5] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011. [Online]. Available: https://doi.org/10.1177/1094342011403516

[6] D. Ajwani, U. Meyer, and V. Osipov, "Improved external memory bfs implementations."

[7] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *2006 International Conference on Parallel Processing (ICPP'06)*, 2006, pp. 523–530.

[8] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ser. STOC '78. New York, NY, USA: Association for Computing Machinery, 1978, p. 114–118. [Online]. Available: https://doi-org.leo.lib.unomaha.edu/10.1145/800133.804339

[9] S. Beamer, "Understanding and improving graph algorithm performance," Ph.D. dissertation, EECS Department, University of California, Berkeley, Sep 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-153.html

[10] D. P. Gregor and A. Lumsdaine, "The parallel bgl : A generic library for distributed graph computations," in *Workshop on Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[11] J. Gilbert, S. Reinhardt, and V. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science and Engineering*, vol. 10, pp. 20–25, 03 2008.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 135–146. [Online]. Available: https://doi-org.leo.lib.unomaha.edu/10.1145/1807167.1807184

[13] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, Aug. 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[14] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[15] "GAP Benchmark Suite Repository," https://github.com/sbeamer/gapbs.

[16] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 293–304. [Online]. Available: https://doi.org/10.1145/3087556.3087580

[17] S. Eyerman, W. Heirman, K. D. Bois, J. B. Fryman, and I. Hur, "Many-core graph workload analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[18] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 775–787. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00068

[19] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 416–427. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00041

[20] V. Young, C. Chou, A. Jaleel, and M. K. Qureshi, "Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 328–339. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00036

[21] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 299–312. [Online]. Available: https://doi.org/10.1145/2967938.2967948

[22] J. D. Leidel, X. Wang, B. Williams, and Y. Chen, "Toward a microarchitecture for efficient execution of irregular applications," *ACM Trans. Parallel Comput.*, vol. 7, no. 4, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3418082

[23] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[24] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, pp. 163 – 177, 2001.

[25] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–8.

[26] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 12–21.

[27] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0196677482900086

[28] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 672–680. [Online]. Available: https://doi-org.leo.lib.unomaha.edu/10.1145/2020408.2020513

[29] U. Meyer and P. Sanders, "Delta-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003, 1998 European Symposium on Algorithms. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403000762

[30] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, "Optimizing ordered graph algorithms with graphit," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 158–170. [Online]. Available: https://doi-org.leo.lib.unomaha.edu/10.1145/3368826.3377909

[31] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi-org.lib-e2.lib.ttu.edu/10.1145/2020373.2020375

[32] "OpenSHMEM Specification." [Online]. Available: http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf

[33] P. Erd˝os and A. R´eyni, "On random graphs. i." *Publicationes Mathematicae*, vol. 6:290–297, 1959.

[34] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *Knowledge Discovery in Databases: PKDD 2005*, A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 133–145.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

[35] "Graph500 benchmark," http://www.diag.uniroma1.it//challenge9/, 2006.
[36] X. Wang, J. D. Leidel, B. Williams, A. Ehret, M. Mark, M. A. Kinsy, and Y. Chen, "xbgas: A global address space extension on risc-v for high performance computing," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 454–463.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE