A Scale-Free Structure for Real World Networks

Richard M. Veras and Franz Franchetti
Department of Electrical and
Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
Email: {rveras, franzf}@cmu.edu

***Abstract*—The field of High Performance Computing (HPC) is defined by application in physics and engineering. These problems drove the development of libraries such as LAPACK, which cast their performance in terms of more specialized building block such as the BLAS. Now that we see a rise in simulation and computational analysis in fields such as biology and the social sciences, how do we leverage existing HPC approaches to these domains. The GraphBLAS project reconciles graph analytics with the machinery of linear algebra libraries. Like their Dense Linear Algebra (DLA) counterpart, the GraphBLAS expresses complex operations in terms of smaller primitives.**

This paper focuses on efficiently storing real world networks, such that for these graph primitives we can obtain the level of performance seen in DLA. We provide a hierarchical data structured called GERMV, which is an extension of our previous Recursive Matrix Vector (RMV). If the network in question exhibits a scale-free structure, namely hierarchical communities, then our data structure enables high performance. We demonstrate high performance for Sparse Matrix Vector (spMV) and PageRank on real world web graphs.

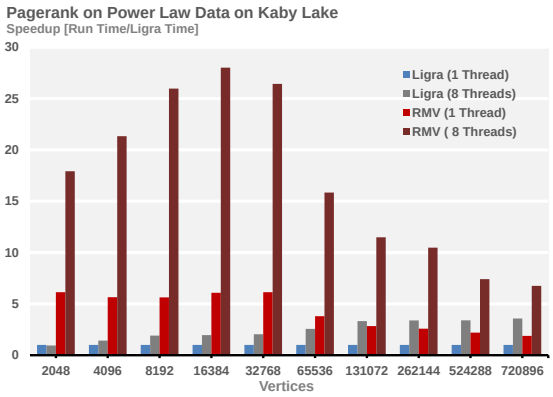

Fig. 1. Our implementation of PageRank, using our GERMV data structure, outperforms Ligra on web graphs of various sizes. Our data structure and algorithm allow us to effectively use the data cache for operations over real world graphs. The key is in identifying the hierarchical clusters in the dataset, storing them in an efficient structure, and computing on them in a cache-aware manner.

I. Introduction

In this article, we extend our previous work in [1] on Sparse Matrix-Vector (spMV) computations over synthetic scale-free networks to graph operations over real-world scale-free networks. We bridge the prior work by identifying hierarchical structures in real-world graphs that allow us to map the graph data to the memory hierarchy. We take advantage of this structure using our hierarchical storage format. This structure allows us to efficiently compute spMV and spMV-like operations on modern machines with deep cache topologies. In Figure 1, we put our approach to the test for the PageRank algorithm on web data. Our implementation outperforms Ligra – the state of the art for shared memory systems.

The contribution of this work are as follows:

- Clusters in scale-free graphs are important for information flow, therefore we show how to hierarchically partition real world graphs using their clusters.
- We capture these clusters in our hierarchical data structure, GERMV, which allows efficient access and manipulation of these clusters.
- We demonstrate how to obtain performance, using these clusters stored in our data structure, in our small GraphBLAS-like library.

II. Background

The graph computation story is rapidly unfolding, and we can classify the emerging work into two distinct, but not necessarily incompatible approaches: the data flow approach and the linear algebra-like approach.

In the data flow approach, the graph frameworks decouple the computation from the data access pattern. Graph operations are implemented in terms of small atomic functions over vertices and edges, and these functions are executed when their inputs have been modified. This process either occurs as bulk computations or as the new data is made available.This process continues until no more vertices or edges are modified. Prime examples of this approach include Pregel [2], Graphlab [3], Ligra [4]. There are a myriad of optimizations to this approach, for example GraphChi [5], X-Stream [6] reshape the graph to take advantage of the memory hierarchy. A more aggressive optimization seen in Galois [7], and its extension in Elixir [8], introduce the notion of speculation by requiring the atomic functions be invertible. Many paths in the graph are explored speculatively, and if a path is no longer viable, its work is undone by applying the inverse of the sequence of instructions that led to it.

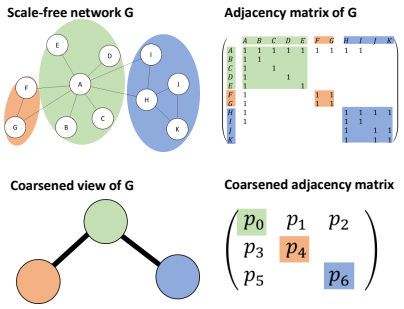

Fig. 2. In the top left, we illustrate an example of a scale-free graph. The key features are the hub vertices and the cluster communities surrounding them. We highlight these clusters in orange, green and blue. On the top right, we show the corresponding adjacency matrix to the graph. Note that we highlight the elements that correspond to the clusters. On the bottom left, we show a coarsened view of the graph, where we replace the clusters with super-nodes. To the right of that we show the corresponding adjacency matrix. Our GERMV data structure captures the structure of the graph with multiple coarsened views of the dataset, in a manner that allows efficient access to these clusters.

For the linear algebra approach to graph analytics, we have the Combinatorial BLAS [9] and Knowledge Discovery Toolbox [10], which represent graph operations as matrix operations over specialized semirings. These semirings capture the desired graph computation by replacing the addition and multiplication operator with functions fitting the graph operations. These libraries leverage many of the approaches used for sparse matrix computations on distributed memory systems and reduce the burden of obtaining performance on the programmer. This approach is made into an API in the GraphBLAS project [11] through the use of C++ templates. The goal of the GraphBLAS is to define an interface for common graph analytic routines that can be tuned to the target system. The result is that algorithms for these routines are implemented as iterative sparse matrix vector products on user defined semirings.

Hierarchical Clustering in Real-World Graphs. The main assumption, on which we base our data structure, is that real-world scale-free networks contain a hierarchical clustering behavior, and that information flow through these graphs occur predominately within these clusters. We use this assumption to construct a data structure that captures this behavior, and we compute on this structure in a way that maximizes cache reuse of these clusters. Prior work has observed this key property, that in real-world networks there is the hierarchical organization of clusters [12], [13], [14]. Vertices in these graphs hierarchically organize to form a recursive cluster structure. We can visualize this with the hub and spoke model. The vast majority of vertices have only a few edges (spokes) that connect to vertices with many edges (hubs). If we were to collapse the hubs and their spokes into a single *super-node* we would see the exact same hub and spoke pattern over these super-nodes. Thus, we can repeat this pattern until there is a single super-node. We illustrate this in Figure 2.

A consequence of this hierarchical topology, hubs are critical for information flow into their communities – a notion reinforced by observations in real-world networks. For example, in [15] the author observed that in infection models over scale-free graph, highly connected hubs contract and spread the infection very quickly. Thus, providing preferential treatment to highly connected hubs decrease the overall infection rate. Similarly, the authors in [16] observed that for viruses spreading on computer networks, that topology, not the spreading rate of the virus, determines the overall rate in which machines are infected. In [17], the authors provided another perspective of information flow through scale-free topologies. They observed that information originating from a community is more important within the community than to vertices outside of the community. This is explained by the fact that, in scale-free graphs, vertices within a community have high connectivity with each other, but very little connectivity outside. Communities in real-world scale-free networks are hierarchically clustered. Information flowing throughout the graph travels through highly connected hubs, but most information within a community flows locally. We exploit this by storing the graph in a manner that preserves this hierarchical structure and matches it to the cache hierarchy. In the next sections, we describe the graph operations we target, the algorithms selected in their implementation, and the optimizations we perform. We take advantage of the hierarchical to extract performance at each step of the way.

Operations over Graphs. In this article, we target Sparse Matrix Vector Product (spMV) and the PageRank operation. The spMV operation computes $y = Ax$, where y and x are dense vectors and A is a sparse matrix. This operation serves as a proxy for more complex operations and has highly tuned implementations on modern hardware. We will demonstrate how graph algorithms can be built on spMV, and why it is important that this operation is efficient. Our second operation, PageRank [18], determines the importance of website based on the probability that a random web surfer will reach that page. In this GraphBLAS style, we implement this as an iterative sparse Matrix-Vector product where for each vertex we compute the weighted sum of all incoming edges. Between every iteration the weights of each vertex are dampened and a fudge factor is added. This continues until the difference between two iterations is less than a given threshold.

Graph Operations as Matrix Operations. In this chapter, we implement graph operations using the approach in the GraphBLAS [11]. In this approach, graph operations are represented as transformations of a module defined over a particular semi-ring that is selected for the operation.

Spacial Blocking for Graphs. We want to capture clusters hierarchically in blocks that fit the memory hierarchy. This builds on the assumption that information flows within scale-free graphs and leverages the fact that nearby vertices are more likely to communicate with each other than distant neighbors. By capturing the proximity of this vertices in cache, we can insure that communication between them is efficient. Thus, if two vertices are neighbors in the graph then they will be neighbors in memory and utilize spacial locality.

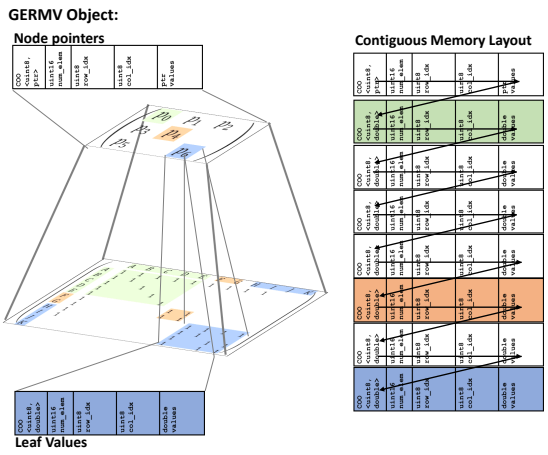

Fig. 3. In this figure, we show our GERMV object containing the graph from Figure 2. The top level represents a coarse view of the graph, and is stored as a sparse matrix container of pointers. The bottom tier contains blocks of elements from graph, and is stored as a series of sparse matrix containers. On the right, we show how these sparse matrix containers can be stored contiguously in memory. This allows efficient computation over the data.

III. Mechanism

In the previous section, we discussed the idea that real-world scale-free networks exhibit a hierarchical cluster structure that we can use to hierarchically partition a graph. We described a method for representing graph algorithms in terms of linear algebra-like operations. Lastly, we described how to use the hierarchical partitioning for space and time tiling on these linear algebra-like algorithms. In this section, we describe how we go from this theory – hierarchical partitioning, linear algebra representation and space/time tiling – to a high performance implementation of a graph library.

We build our graph library around a generalized version of the Recursive Matrix Vector (RMV) data structure that we presented in [1]. This structure allows us to capture the hierarchical recursive structure of the graph and traverse through that structure efficiently. We then develop recursive graph algorithms that operate on this structure. These algorithms algorithms divide the graph into very small subgraphs, which are computed on using extremely efficient, automatically generated kernels.

Generalized Hierarchical Sparse Framework. In our previous work [1], we developed a hierarchical sparse matrix framework for scale-free networks. This implementation was optimized for Kronecker Graphs [19], a class of scale-free networks. This data structure stores structured Kronecker Graphs hierarchically and efficiently in memory. While Kronecker Graphs can approximate real-world networks, more flexibility is needed for storing real-world networks. For this flexibility, we generalize our RMV data structure to accommodate real-world data, while retaining the hierarchical sparse structure. We call this format Generic Recursive Matrix Vector Storage (GERMV), and we illustrate this data structure in Figure 3.

Like the RMV data structure, GERMV is a tree-like hierarchical matrix storage. It stores a matrix as a hierarchical nesting of blocks. Further, each of these blocks is described by a node. This structure is very similar to KD-trees [20] if each dimension could be partitioned more than once. Alternatively, this structure can be viewed as Hierarchically Tiled Array [21] or FLASH [22] if the base data type in each tile could be sparse instead of dense.

The key difference between the RMV data structure and the GERMV data structure is the generalization of indexing. In the original RMV structure, indexing is restricted to a bit-matrix, but in the GERMV indexing can be done in COO, CSR, bit matrix, or a dense matrix format. To accommodate generic indexing and generic data container in C, we break up the GERMV container into two pieces, a base container that determines the payload and a payload that contains the indexing and data elements. We illustrate this structure in Figure 4 and in the following listing we show the base container.

```
typedef struct germv_base_ts
{
  enum  type;
  void *payload
}
```

The field *type* determines how the following field payload needs to be interpreted. Essentially, this is one method for implementing class-like structures in C.

```
typedef struct payload_dense_double
{
  double [][] data;
}

typedef struct payload_coo_uint16
{
  uint8_t row_idx;
  uint8_t col_idx;
  uint16_t   data;
}
```

In this code snippet, we demonstrate how to capture different formats as wrappers. If the base object's type is *dense double* then its payload is cast using the appropriate wrapper. This approach to classes provides us with a low overhead method of implementing an abstract hierarchical matrix type. In the next section, we demonstrate how we use GERMV object.

Abstract Matrix Types. The GERMV structure allows us to capture a graph as a hierarchical partitioning. This is done by creating a tree-like representation of the matrix where each level of the tree represents a partitioning of the parent node. To accommodate this, the GERMV allows for arbitrary data types. Typically, the leaf blocks will contain a value data type (i.e. float or int), whereas the interior nodes - or hierarchically blocks - have values with a pointer data type. These pointers, in turn, point to their child blocks. We can continue this recursively until we hit the leaf nodes. For example, in the following listing we describe a payload type that indexes using the COO format:

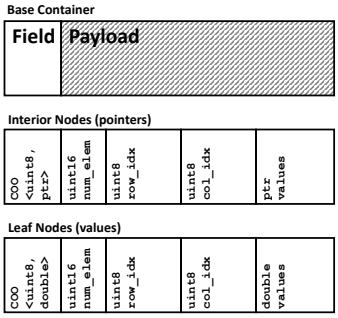

Fig. 4. Every item in our GERMV object is stored in class-like containers. The header of this container signifies the type of data that follows. This method for encoding the data allows us to pack the graph in a contiguous stream from memory.

```
typedef struct payload_coo_germv_base
{
  uint8_t      row_idx;
  uint8_t      col_idx;
  germv_base_t data;
}
```

Given this pointer data type, we can create a hierarchically partitioned graph with a single vertex and edge, one leaf node and one interior root node.

```
#define CONSTRUCT(type) ...
#define ADD_ELEM(package,i,j,data) ..

germv_base_t *interior =
  CONSTRUCT( payload_coo_germv_base );
germv_base_t *leaf     =
  CONSTRUCT( payload_coo_float );

ADD_ELEM(interior, 0,0, leaf);
ADD_ELEM(leaf,     0,0, 3.14f );
```

We create two GERMV objects, one is the root or interior node and the other is the leaf node. We use a pointer data type with COO indexing for payload type for the root node. For the leaf node we use COO indexing with a floating point indexing. Once the objects are constructed we attach the leaf to the interior by adding it as an element and similarly we add an edge and value to the leaf using the same mechanism.

A High Performance spMV Implementation. Now that we have a hierarchical data structure, we can build recursive algorithms for the operations we are interested in. We start with Sparse Matrix Vector Product (spMV) because it is an important operation on its own and because it will form the basis of the graph operations that we are ultimately interested in. The spMV computes $y = Ax$ where $y \in \mathbb{R}^m$, $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m \times n}$. This operations requires mn multiplications and additions and at the very least $2m+n+mn$ memory accesses. The latter figure assumes perfect reuse of the y and x vector. The cost of memory access is typically more expensive than computation, therefore we want to maximize reuse. In order to do this, we construct a recursive spMV over our GERMV and we match the block sizes – really cluster sizes – to fit the cache hierarchy. After wWe divide the GERMV matrix into blocks, then recursively apply these spMV algorithms until we reach the base elements of the matrix. We realize this process in the following code snippet:

```
spmv_dispatch(A,x,y)
  {
    if( is_leaf(A) )
      spmv_kernel(A,x,y)
    else if( is_node(A) )
      spmv_block_row_and_col(A,x,y)
  }
```

The algorithm first determines if we are dealing with a leaf GERMV container (an object with real values) or a node GERMV container (an object with pointer values), then dispatches to the appropriate code. If the GERMV is a node, then spMV is computed recursively on the non-NULL pointer values by calling the *dispatch* function on the value. If the GERMV is a leaf, then it is computed.

```
spmv_block_row_and_col(A,x,y)
{
  for( i = 0 .. mb-1 )
    for( j = 0 .. nb-1 )
      Ab = get_elem( A, i,j );
      yb = get_elem( y, i );
      xb = get_elem( x, j );
      spmv_dispatch( Ab,xb,yb )
}
```

In this listing we show a blocked spMV algorithm that partitions both the rows and columns. The values m_b and n_b correspond to the number of blocked rows and columns, respectively. If we effectively capture the clusters of the graph in our GERMV object, then these partitions correspond to the clusters in the graph. This insures that we are operating on the graph clusters within the cache. In the next listing, we show the base case for our spMV.

```
spmv_kernel(A,x,y)
{
  for( i = 0 .. mb-1 )
    for( j = 0 .. nb-1 )
      y[i] += A[i][j] * x[j]
}
```

To summarize our spMV implementation, we divide the operation by recursive applications of a blocked row and column spMV algorithm. The block sizes for each recursion match the size of the cache at each level of the hierarchy. Further, the partitioning of the spMV operation and the block sizes determine how the matrix is recursively stored in our GERMV object. Assuming that the vertices in the adjacency matrix are ordered in locality preserving manner, then our approach keeps graph cluster in cache and effectively uses that locality.

IV. Analysis

For our experiments, we use a graph from the Stanford WebBase [23]. The graph is a web crawl of Berkeley and Stanford websites called *web-BerkStan*. Each vertex represents a page and an edge represents a hyperlink from one page to

the other. The graph itself has $685,230$ vertices and $7,600,595$ edges.

In order to show a trend over a range of graph sizes with same overall graph behavior, we created ten graphs from the original web-BerkStan, $G = (V, E)$. We label all of the vertices $v_i \in V$, where $0 \leq i < |V|$ and create the ten sub graphs from $G_i = (V_i, E_i)$, where $V_i = v_j \in V | j \leq i$ and $E_i = E \cap V_i \times V_i$. By using sub-graphs of a range of sizes we can examine the behavior of our library as the graph size changes, but the underlying structure stays the same.

Depth	Row Block Size (m_b)	Col Block Size (n_b)
1	720896	720896
2	65536	65536
3	16384	16384
4	4096	4096
5	1024	1024
6	256	256

Fig. 5. These are the GERMV data structure blocking dimensions used in our experiments.

Storing the Graph in the GERMV Object. For all of the experiments, we pack the target graph in a GERMV object and perform the spMV and PageRank operations over this packed object. The partition size and number of partitions is dependent on the organization of the cache hierarchy and the size of each cache. For example, in the graph G_{720896} we partition and store the graph hierarchically according to the blocking dimensions listed in Figure 5. To illustrate how the graph is captured, we show views of the adjacency matrix at various granularity of the GERMV object in Figure 6. At each depth of this table we provide spy-plot of the non-zero blocks at that depth, along with a histogram of the density of each of those blocks. With the exception of depth $d = 6$, each graph of depth $d = n$ corresponds to a coarsening of the graph of depth $d = n+1$. By starting with a graph where the elements are labeled in an order that maintains the cluster structure of the graph at depth $d = 6$, we can maintain that clustering at each level of $d < 6$ through coarsening. Two key observations are that the distribution of block densities follows a power law distribution, and the non-zero blocks are fairly dense despite the graph being very sparse.

Overall Performance. In the top row Figure 7, we show the results for our overall multi-threaded performance experiment. In this experiment, we compare our performance against the CSB [24] and MKL's COO implementation of spMV. The MKL COO implementation is selected as reference implementation and the CSB implementation is the current state-of-the-art. On both machines, we use the maximum number of threads available. Our spMV GERMV efficiently uses the system cache, multiple threads and the memory bandwidth. In both results, our performance is significantly greater than CSB. In particular on the Kaby Lake (Top Left) when the vector is small enough to fit in the cache the performance is 30% greater than CSB. This we attribute to our efficient use of the data cache. It is worth noting that the CSB implementation

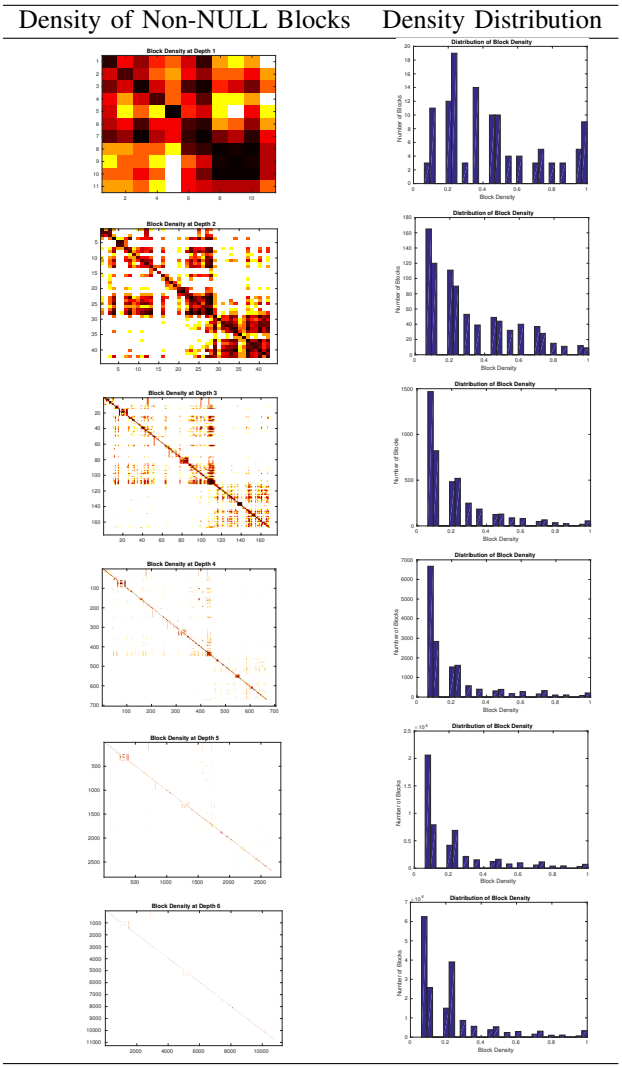

Fig. 6. We store our target data set – web-BerkStan – hierarchically in our GERMV object. In this table we show the density of the data stored at each depth of this hierarchy. On the **left** we provide a heatmap that shows the density of each block, and on the **right** we have a histogram comparing the densities of these blocks.

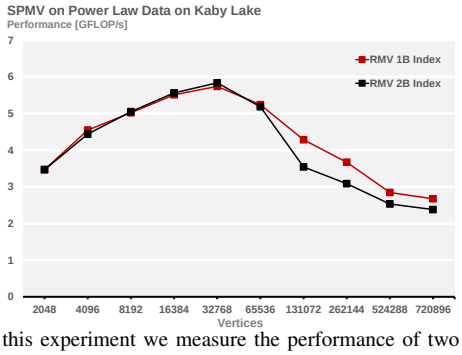

Fig. 8. In this experiment we measure the performance of two implementations using different data widths (1 and 2 Bytes) for storing indexing data.

was not designed for NUMA systems, which is why on the Xeon our performance is substantially higher than CSB.

Parallel Scaling. In the experiments shown in the bottom

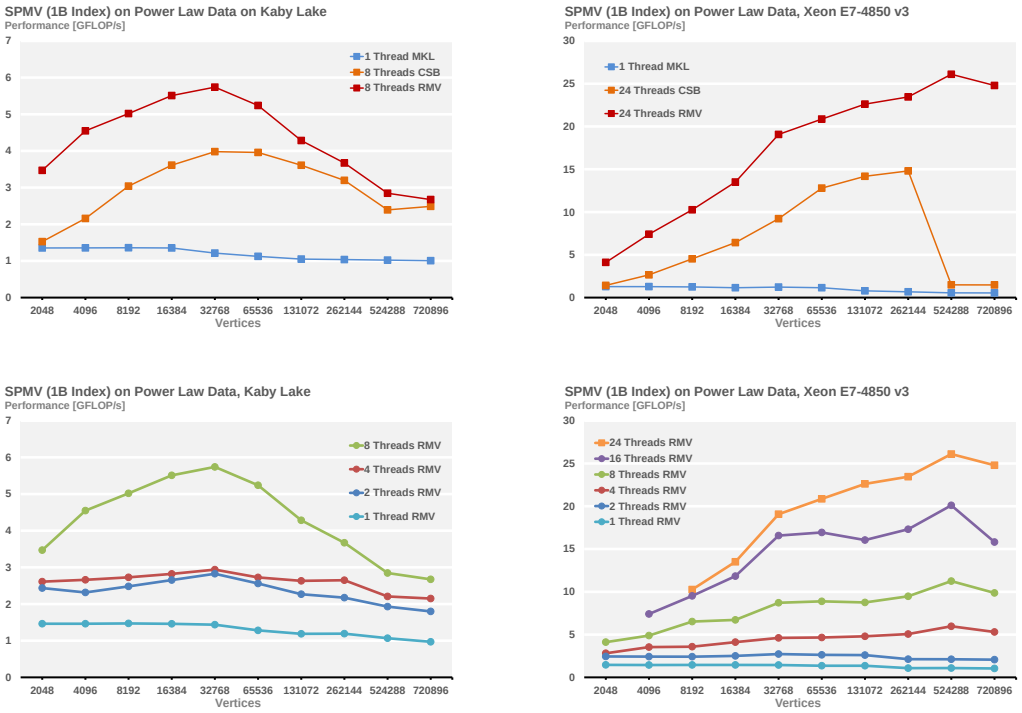

Fig. 7. **Top:** We compare the performance of our spMV implementations against CSB and MKL's COO implementation for various sized subgraphs of a web graph. We do this on two different machines. **Bottom:** We evaluate the performance scaling of our implementation as we increase thread count.

row Figure 7, we compare the performance of our GERMV spMV implementation for various thread counts. The idea is show that our implementation scales, as we add additional threads. On the Kaby Lake (Bottom Left) we can divide the results into two parts, in cache and out of cache. For the out of cache behavior our speedup is linear with respect to the number of threads. When the problem size fits in the cache, we get super linear speedup because as we add threads our implementation has access to more cache and therefore can have greater reuse. On the Xeon (Bottom Right) we see a more linear speedup as we increase the thread count. The takeaway is that our implementation scales across multiple threads and multiple sockets.

Compact Indexing. Our GERMV implementation allows us to use arbitrary indexing format. This feature allows us to use an indexing format that minimizes the amount of overhead needed for indexing. Because the indices we use will always be smaller than the block sizes selected, we only need to use enough bytes to encode those indices. In the experiment in Figure 8, we compare the use of 1 Byte indexing versus 2 Bytes for indexing. When the number of vertices fit in the cache there is no difference in performance. However, the moment our implementation needs to access main memory the extra overhead reduces performance.

V. Summary

In this article, we extended our previous work on a scale-free data structure (RMV) and generalized it to real world graphs using our GERMV object. We found that we could use the hierarchical clustering behavior of real world graphs to guide how we partition the graphs, which in turn determined how we transformed the data layout. This gives us the benefit that neighboring vertices, which communicated frequently with each other, are cache adjacent. Because these small sub-graphs are cache resident, we can use tuned kernels to process them. By combining these two pieces, efficient layout and kernels, our spMV and PageRank outperform the state of the art. Our implementation assumes the input graph is efficiently ordered such that the diagonal blocks represent clustered communities. For the dataset we tested, the ordering was sufficient. However, we can improve the ordering using graph partitioning at the expense of an additional preprocessing step. For future work we want to focus on how the data is collected and how we can modify this step to improve the ordering of the resulting data. The rationale is to match the traversal of the desired graph algorithm to the traversal. This integrates the data layout transformation in the collection and could insure that data arrives and is stored in the order that it will be computed.

Acknowledgment

This work is supported by the Department of Defense under Contract No FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

REFERENCES

[1] R. Veras, T. M. Low, and F. Franchetti, "A scale-free structure for power-law graphs," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7.

[2] 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*. ACM, 2010, pp. 135–146.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[4] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[5] A. Kyrola, G. E. Blelloch, C. Guestrin *et al.*, "Graphchi: Large-scale graph computation on just a pc." in *OSDI*, vol. 12, 2012, pp. 31–46.

[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.

[7] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 12–25.

[8] D. Prountzos, R. Manevich, and K. Pingali, "Elixir: A system for synthesizing concurrent graph programs," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 375–394, 2012.

[9] 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.

[10] A. Lugowski, A. Buluç, J. R. Gilbert, and S. Reinhardt, "Scalable complex graph analysis with the knowledge discovery toolbox," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE, 2012, pp. 5345–5348.

[11] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–9.

[12] E. Ravasz and A.-L. Barabási, "Hierarchical organization in complex networks," *Physical Review E*, vol. 67, no. 2, p. 026112, 2003.

[13] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, vol. 466, no. 7307, pp. 761–764, 2010.

[14] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, and A.-L. Barabási, "Hierarchical organization of modularity in metabolic networks," *science*, vol. 297, no. 5586, pp. 1551–1555, 2002.

[15] M. Barthélemy, A. Barrat, R. Pastor-Satorras, and A. Vespignani, "Velocity and hierarchical spread of epidemic outbreaks in scale-free networks," *Physical Review Letters*, vol. 92, no. 17, p. 178701, 2004.

[16] R. Pastor-Satorras and A. Vespignani, "Epidemic spreading in scale-free networks," *Physical review letters*, vol. 86, no. 14, p. 3200, 2001.

[17] F. Wu, B. A. Huberman, L. A. Adamic, and J. R. Tyler, "Information flow in social groups," *Physica A: Statistical Mechanics and its Applications*, vol. 337, no. 1, pp. 327–335, 2004.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[19] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, March 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039

[20] S. Wess, K.-D. Althoff, and G. Derwand, "Using kd trees to improve the retrieval step in case-based reasoning," in *European Workshop on Case-Based Reasoning*. Springer, 1993, pp. 167–181.

[21] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzaran, G. Almasi, J. Moreira, and D. Padua, "The hierarchically tiled arrays programming approach," in *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*. ACM, 2004, pp. 1–12.

[22] T. M. Low and R. van de Geijn, "An API for manipulating matrices stored by blocks," Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-2004-15, May 2004.

[23] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley, "Stanford webbase components and applications," *ACM Transactions on Internet Technology (TOIT)*, vol. 6, no. 2, pp. 153–186, 2006.

[24] A. Bulu, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *IN SPAA*, 2009, pp. 233–244.