SAND2018-2105C

Experimental Design of Work Chunking for Graph Algorithms on High Bandwidth Memory Architectures

George M. Slota
Rensselaer Polytechnic Institute
Troy, NY
slotag@rpi.edu

Sivasankaran Rajamanickam
Sandia National Laboratories
Albuquerque, NM
srajama@sandia.gov

***Abstract*—High Bandwidth Memory (HBM) is an additional memory layer between DDR and cache, and it currently exists in the form of Multi-Channel DRAM (MCDRAM) on the Intel Knight's Landing manycore architecture. Its purpose is to increase available memory bandwidth to maximize processor throughput. This work explores optimizing the label propagation community detection algorithm on the KNL, as this algorithm and its variants find broad usage in community detection. This algorithm's processing pattern also represents broader class of vertex-centric programs. As HBM becomes more common in new HPC systems, it is important to determine how best to exploit this memory layer for memory-starved graph and combinatorial algorithms. This work experimentally examines breaking up the algorithmic work into HBM-resident chunks, along with a parametric study of associated variations and optimizations. In general, we find our chunking methodology does not harm solution quality and can improve time to solution for label propagation. We believe these results would likely generalize to other vertex-centric algorithms as well.**

***Index Terms*—graph processing; label propagation; community detection; Intel Knight's Landing**

I. Introduction

An ongoing hardware trend for high performance computing (HPC) architectures is the continuing expansion of the storage, memory, and compute hierarchy. Computation throughput decreases with the distance data is from the core, and advanced caching algorithms and expanding cache hierarchies have allowed continued performance gains until recently. Alternative strategies are now being more broadly considered; one such strategy is a multi-level memory approach containing a "high bandwidth memory" (HBM) layer between DDR and cache. Leadership-class compute platforms such as the Trinity Phase 2 system at Los Alamos contain Intel Knight's Landing (KNL) processors are equipped with such a memory layer. As such, research efforts are underway to develop software strategies which effectively use this HBM layer [1].

In particular, one area that might be able to effectively utilize HBM is in parallel software implementations for graph analytical algorithms. These algorithms are typical memory-bounded, with memory bandwidth and latency being key performance limiters [16], [23], [32]. This is in contrast to more typical HPC scientific applications, which have a considerably higher ratio of compute-to-memory access. Further, while HBM on the KNL can naïvely be used as an additional cache layer without explicit application developer input [27]. However, this usage can be suboptimal for data-heavy algorithms processing large irregular graphs – memory access patterns in these instances are too irregular and over too large of a memory footprint for predictive HBM caching to be effective. We observe HBM in cache mode to be most performant for the regular and streaming memory accesses common to more traditional scientific computing codes. By developing methods to effectively utilize this high bandwidth layer, there is the potential for the considerable acceleration of memory-bounded graph applications relative to compute-bounded codes.

In this paper, we describe **chunking**, where the total algorithmic work of an iterative graph computation is broken up into discrete HBM-resident *chunks*. Each of these chunks are iteratively brought into and processed within HBM until a final solution of desired quality is produced. While the general idea of work chunking isn't new (.e.g, memory-limited devices such as GPUs or single-node CPU systems processing an SSD-stored large graph have used similar approaches), its application towards a relatively novel upcoming architecture is a promising research area.

As our prototypical algorithm, we select the label propagation community detection algorithm [21]. This algorithm is an excellent choice to evaluate our methods for two reasons. Firstly, in addition to its wide utilization for community detection and similar problems, the vertex-centric [17], [18] nature of this algorithm makes it representative of a broad class of general graph computations. Secondly, the output of the algorithm is nondeterministic by design and the solution quality can vary based on processing methodology. These two factors enable us to explore a variety of techniques for breaking up the computation within the HBM layer while evaluating the impact of these techniques on output quality and time to solution. In addition, by following a general vertex-centric algorithmic pattern, our optimizations and observations might be generalizable to other vertex-centric graph algorithms in both memory-constrained and distributed environments.

A. Contributions

This experimental study has several contributions:

1) We develop a scalable and generalizable approach for efficiently running parallel vertex-centric graph algorithms in multilevel memory.
2) We demonstrate a proof-of-concept of our approach by implementing the label propagation community detection algorithm and showing its speedup relative to naïve methods.
3) We demonstrate that we can adjust the parameters controlling our approach such that it does not result in degradation of (in some instances improve) solution quality.

The end goal of this work is to serve as a guide for graph analytic application developers intending to utilize KNL architectures for high performance parallel graph processing.

II. Background

A. Label Propagation

The label propagation algorithm is an efficient linear-time community detection algorithm that was originally proposed a decade ago [21]. Community detection is informally defined as finding dense clusters which have a high ratio of internal to external edges within a large network; though several more formal definitions exist [6]. The label propagation algorithm effectively minimizes inter-community cuts from the local perspective of all vertices, through vertices assigning themselves to a community (label) that appears most often in their neighborhood; ties are broken randomly. Nondeterminism is introduced into the behavior of the algorithm through randomization of the order in which vertices are processed.

While a large number of "competing" community detection algorithms exist, label propagation is desirable for its ease of parallelization and linear execution time. The baseline algorithm has been extended and optimized for a variety of applications, including analysis of protein interaction networks [8], Facebook [22], and many other applications [9], [15]. Optimizations in these work include considering the *history* of label assignments, considering multiple label assignment possibilities per vertex (including overlapping communities), and modifications to the optimization function. Label propagation has additionally been extended to the related problem of balanced graph partitioning in several work [19], [25], [29], and a related approach is often used for general semi-supervised learning problems [33]. Its vertex-centric approach leads to straightforward and efficient parallelization in shared-memory, distributed-memory on up to trillion-edge graphs [26], and even streaming or dynamic environments [31].

A note on solution quality: While there exists a general consensus on the importance of identifying communities within a graph for several applications, actually defining solution quality for community detection is a continuing debate. For datasets where there exists no ground truth, modularity and conductance are often the primary optimization targets [6], [14] – though it is established that modularity maximization is unable to capture smaller-sized communities [7]. When a *ground truth* exists, often-used measure include normalized mutual information (NMI) and the rand index, among others. For this work, we consider the popular metrics of modularity, rand index, and NMI for evaluating solution quality.

Although, we stress specifically that our goal of our work is not to optimize the label propagation algorithm for a specific dataset or problem – there already exists a substantial body of work on variants of the basic label propagation algorithm, as we've noted. Rather, our goal in this work is to develop an approach for accelerating label propagation within a multi-level memory hierarchy where solution quality relative to the baseline approach is not degraded. The general approaches we discuss should be broadly applicable to all specialty label propagation variants as well as extendable to vertex-centric algorithms as a whole.

B. High Bandwidth Memory

High bandwidth memory (HBM) in the form of Multi-channel DRAM (MCDRAM) was introduced by Intel for their Knight's Landing (KNL) manycore processor architecture. The primary purpose of this HBM layer is to address the fact that DDR4 is unable to supply the bandwidth necessary to maximize processor throughput; on the Stream Triad Benchmark, KNL's eight MCDRAM channels can supply approximately 450 GB per second aggregate bandwidth while its six DDR4 channels can supply approximately 90 GB per second [28].

As a representative system of current and potential future multi-level or HBM-based platforms, we run on the *Bowman* testbed cluster at Sandia National Labs. This system contains nodes with manycore Knight's Landing (KNL) processors and 16GB MCDRAM along with 96GB DDR4. We have developed an internal library – `libtlvl` (lib two-level) – that allows us to directly allocate to HBM via the `memkind` library [4]. We have also implemented an allocator object for use with Standard Template Library (STL) containers such as vectors and maps; this enables these containers to call our `libtlvl` functions to allocate, deallocate, and resize arrays directly in HBM. We use the quadrant clustering mode of KNL. The HBM on a KNL can run in multiple "modes"; we describe and consider three of them in our testing.

- *Flat* – flat MCDRAM mode, which requires the user to explicitly control memory allocation, deallocation, and potential transfers between DRR and HBM. In this mode, no HBM is utilized as cache.
- *Cache* – 100% cache mode, in which the KNL automatically uses the HBM as an extra layer of cache. This mode does not require any explicit user interaction or input to use the HBM.
- *Hybrid* – hybrid 50/50 MCDRAM and cache mode, which is effectively a cross between *Flat* and *Cache* modes; half of the available HBM is treated as cache while the other half is available for explicit allocation by the user.

III. Multilevel Memory Label Propagation

As mentioned, our primary approach for multilevel memory label propagation is based on an approach of work *chunking*. The general idea behind chunking is to take a portion of some iterative problem, bring it to HBM, work on it for some number of iterations, update some global state, then remove it from HBM and retrieve some new portion of the problem to work on. For an iterative graph algorithm such as label propagation, we could do multiple *local* iterations of work on each chunk in succession, along with some number of *global* iterations until convergence is reached. This is represented for label propagation in Algorithm 1. While the idea is simplistic in general, there are several nuances that make engineering a chunking-based solution difficult in practice.

Algorithm 1 Baseline Multilevel Label Propagation Algorithm.

```
1:  procedure CHUNKING(G(V, E), C_num, C_iter)
2:      for all v ∈ V do
3:          L(v) ← id(v)
4:      updates ← 1
5:      while updates ≠ 0 do
6:          updates ← 0
7:          for c = 0 ··· (C_num − 1) do
8:              V_c ← Chunk(c, V)        ▷ V_c is disjoint chunk c of V
9:              E_c ← ⟨v, u⟩ ∈ E : v or u ∈ V_c
10:             iter ← 0, chunkUpdates ← 1
11:             while iter < C_iter and chunkUpdates ≠ 0 do
12:                 chunkUpdates ← 0
13:                 for all v ∈ V_c do in parallel    ▷ Random Order
14:                     Counts ← ∅
15:                     for all ⟨v, u⟩ ∈ E_c do
16:                         Counts(L(u)) ← Counts(L(u)) + 1
17:                     NewLabel ← Max(Counts(...))
18:                     if NewLabel ≠ L(v) then
19:                         L(v) ← NewLabel
20:                         chunkUpdates ← chunkUpdates + 1
21:                 updates ← updates + chunkUpdates
22:                 iter ← iter + 1
23:     return L
```

The primary variables that determine the speed and quality of a final solution are related to the number of the chunks (C_{num}) and the number of local iterations (C_{iter}); running the inner loop to convergence instead of a fixed number of iterations may very well not produce a globally optimal solution or be efficient in practice. Additionally, the portion of the graph that each chunk represents can also have a large impact on the required amount of work needed for solution convergence. Varying the portions of the graph contained within each chunk through either random allocation, vertex block allocation, edge block allocation, or explicit vertex and edge balanced partitioning are all possible approaches. With each of these variables, one must consider how to handle HBM allocation as well as the explicit storage and transfer of graph and associated vertex state data.

We evaluate all of these variables experimentally in our results. In the following subsection, we describe the several implementation variants we use to examine the various possible impacts of using HBM and work chunking.

A. Method Variants

We evaluate several primary chunking variants that consider the utilization of HBM in the previously described modes (*Flat* – explicit allocation, *Cache* – cache mode, *Hybrid* – 50/50 allocation/cache). Note that the various data structures we're considering for HBM storage include label assignments ($O(n)$ total), the graph structure ($O(m+n)$ total), and thread-owned hash tables ($O(t \cdot d_{max})$ total). n is the number of vertices, m is the number of edges, t is the number of processing threads, and d_{max} is the maximum degree of the graph. The scale of the graphs we experiment on is approximately 2 billion (undirected) edges. Compressed sparse row (CSR) representations of these graphs require approximately $2 \times 10^9 \times 2 \times 4 \approx 16$ GB of memory to store, which is the capacity of HBM on *Bowman*. Including vertex state data, thread-owned data, and potential memory fragmentation, we note that it is not possible to directly allocate a graph of this scale and its associated algorithmic data in HBM.

We have the following method variants for label propagation:

a) Baseline-Cache: This is a baseline label propagation algorithm used for comparison. This is a parallel implementation where work is broken up among threads which, on each iteration, update labels for some subset of vertices. We use standard parallelization practices with OpenMP, with `schedule(guided)` scheduling for threads. Each thread owns a hash table to track *Counts*, where the hash table used in our implementation is `stl::unordered_map`. All allocations are done in DDR. As we ignore allocation in HBM, this variant is only memory-restricted by the size of DDR. Running this variant in *Cache* mode is what we consider as our *true* baseline benchmark, as the choices for this implementation are all reasonable approximations to those that an application developer would make.

b) Baseline-Hybrid: This variant allocates the graph structure and label assignments in DDR, but has the thread-owned hash tables in HBM. We require only $O(t \cdot d_{max})$ space in HBM. We consider this variant as targeted towards running in *Hybrid* mode, where the graph structure and label information might be cached in HBM, but we ensure that the repeatedly-accessed thread-owned data structures are permanently resident in HBM. Note that we have also considered an opposite allocation scheme in which we chunk the graph data and have the thread data be handled by cache. We'll discuss the experimental differences of both in the results.

c) Chunk-HBM: This is our primary chunking variant. We allocate the thread-owned hash tables and the label assignments permanently in HBM. The greatest data cost, the graph structure, is transferred into and out of HBM. We consider both transferring in the graph structure as-needed per-chunk as well as overlapping communication with computation (i.e., we transfer in the next chunk's data as we work on the current chunk – we'll describe our approach in more detail later).

We require $O(n + t \cdot d_{max} + \frac{n+m}{c})$ for this variant, where c is the number of chunks. As irregular graphs in general have $m >> n$, we might hold the assumption that if a graph $O(n + m)$ can be fully resident in DDR, then it is likely we can likely store $O(n)$ label assignments in HBM along with the $O(t \cdot d_{max})$ thread data and $O(\frac{n+m}{c}))$ per-chunk graph data. When double buffering, we require double the storage per-chunk for the graph data.

d) Chunk-HBM-Part: Although we don't explore this variant in our results, we also wish to consider a secondary chunking variant. This variant keeps only the chunk-specific data in HBM. The primary difference is what if we no longer hold the assumption that we have $O(n)$ HBM space, or that we are unable to store the entire label array in memory. This variant would require what is equivalent to a fully distributed graph data structure. Label data for *local* (per-chunk) vertices can be stored in a flat array, while vertices within a one-hop neighborhood of the chunk must have their labels stored in a hash table (as we can't directly access $O(n)$ memory space). On the surface, the cost of this hashing along with the additional transfer costs for the label data make this variant undesirable; however, for future architectures with higher DDR capacity and relatively limited HBM, this variant might be necessary for the processing of next-generation graph data. Even current KNLs can support up to 384 GB DDR.

B. Chunking Partitioning

We also wish to explore exactly *how* to perform potential work chunking, as this might have a large impact on solution convergence and overall processing time. Within our description of chunking, we implicitly describe the partitioning of the graph into vertex-disjoint chunks. We therefore wish to explore some common methods used for the partitioning of a graph within a distributed environment. We utilize the following methods for chunk partitioning:

a) Vertex Block: For this method, chunks are created such that the vertex set within each chunk is approximately the same cardinality. We utilize the vertex identifiers within the original graph and assign vertices to chunks in order of their identifiers. The number of edges is variable among chunks, and can be quite skewed based off of the graph's degree distribution. This method might also benefit from an implicit assumption that the input graph's *natural* ordering of vertices is "good" based on one of the various measures [24]. This assumption can regularly hold, since many large-scale social and web graph datasets are created through some form of crawling, creating a natural breadth-first or depth-first ordering.

b) Edge Block: For this method, chunks are created such that the cardinality of the edge sets for each chunk are approximately equivalent, while the number of vertices can be variable. Similar to vertex block chunking, we create the chunks using the natural ordering of the vertices. To determine these chunks, we can compute prefix sums using the degree of each vertex, and use them to determine offset vertices for the start of each chunk; the number of edges per chunk isn't necessarily exact, but in practice they are the same size within a few percent. This method again can also benefit from a good natural ordering of an input graph. It can additionally benefit from a possibly improved balance when overlapping communication, since the highest transfer cost comes from the number of edges in a chunk.

c) Random: For random chunking, we just randomly assign vertices to chunks. This method of partitioning in general will result in balanced chunks in terms of edges and vertices, but loses the caching locality benefit from a (presumed) well-ordered block method. Additionally, the one-hop neighborhood of each chunk is significantly larger, so if using *Chunk-HBM-Part*, the amount of label information that needs to be hashed correspondingly increases.

d) Explicit: With explicit partitioning, we run the PuLP partitioner [25] on the input graph to compute C_{num} parts with balanced vertices and edges and a minimized edge cut. We note the ideal optimization criteria, at least for minimizing the hashed label information with *Chunk-HBM-Part*, might be communication volume – exploring the benefit of this optimization criteria using other modern partitioners [19] is reserved for future work. The obvious drawback to explicit partitioning is the cost of partitioning (which with PuLP, is essentially performing constrained label propagation) – though this cost can obviously be offset if the part assignment are used through multiple experiments. The benefit of explicit partitioning is that both vertices and edges are balanced per chunk and there might be some additional locality optimization. This method might be especially useful should the input graph be ill-ordered.

C. Overlapping Communication

We explore the overlapping of communication and computation to further optimize our chunking variants. With this approach, some number of threads are portioned off to copy in graph structural data (and vertex state data for *Chunk-HBM-Part*) for the next chunk, while the rest of the threads perform the actual work for the current chunk. The obvious benefit of this approach is hiding the transfer costs at a small cost to threads that compute. The obvious drawback of this approach is the doubling of memory cost due to the additional data structures required to hold the data for the next work chunk. However, empirical observations demonstrate that this approach will generally outperform non-overlapping methods when the increase in required chunks due to memory constraints doesn't greatly increase time to convergence; this is naturally algorithm- and graph-dependent. We show our overlapping algorithm in Algorithm 2.

Note that we dynamically determine the number of threads to allocate for copying and working on each chunk. We start with a single copy thread and increase it depending on the ratio of work time to copy time the chunk. When the chunks are balanced in terms of work, this method can be very effective at finding the optimal assignment of copy and work threads. However, when chunks vary greatly in size, as can be the case with vertex-block chunking, it is better to track *ratio*, t_{copy}, and

Algorithm 2 Overlapping Communication Algorithm

1: **procedure** CHUNKINGOVERLAP($G(V,E), C_{num}, C_{iter}$)
2: **for all** $v \in V$ **do**
3: $L(v) \leftarrow id(v)$
4: $t_{copy} \leftarrow 1$, $t_{max} \leftarrow totalthreads$
5: $t_{work} \leftarrow t_{max} - t_{copy}$
6: *updates* $\leftarrow 1$
7: **while** *updates* $\neq 0$ **do**
8: *updates* $\leftarrow 0$
9: **for** $c = 0 \cdots (C_{num} - 1)$ **do**
10: swap($V_{c_{next}}, V_c$)
11: swap($E_{c_{next}}, E_c$)
12: **Split parallelism into two sections**
▷ E.g: `#pragma omp sections num_threads(2)`
13: **Run this section asynchronously**
▷ E.g: `#pragma omp section`
14: $c_{next} \leftarrow (c+1) \ \% \ C_{num}$
15: $V_{c_{next}}, E_{c_{next}} \leftarrow$ transfer() **with** t_{copy} **threads**
16: *copyTime* $\leftarrow$ timer()
17: `end section`
18: **Run this section asynchronously**
▷ E.g: `#pragma omp section`
19: *iter* $\leftarrow 0$, *ChunkUpdates* $\leftarrow 1$
20: **while** *iter* $< C_{iter}$ **and** *ChunkUpdates* $\neq 0$ **do**
21: *ChunkUpdates* $\leftarrow 0$
22: *iter* $\leftarrow$ *iter* $+ 1$
23: **for all** $v \in V_c$ **do in parallel with** t_{work} **threads**
24: *Counts* $\leftarrow \emptyset$
25: **for all** $\langle v, u \rangle \in E$ **do**
26: *Counts*$(L(u))) \leftarrow$ *Counts*$(L(u))) + 1$
27: *NewLabel* $\leftarrow$ Max(*Counts*$(\ldots)$)
28: **if** *NewLabel* $\neq L(v)$ **then**
29: $L(v) \leftarrow$ *NewLabel*
30: *ChunkUpdates* $\leftarrow$ *ChunkUpdates* $+ 1$
31: *updates* $\leftarrow$ *updates* $+$ *ChunkUpdates*
32: *workTime* $\leftarrow$ timer()
33: `end section`
34: `end sections`
35: $ratio \leftarrow \frac{t_{work} \times workTime}{t_{copy} \times copyTime}$
36: $t_{copy} \leftarrow \frac{t_{max}}{ratio}$
37: **if** $t_{copy} < 1$ **then**
38: $t_{copy} \leftarrow 1$
39: **else if** $t_{copy} > t_{max} - 1$ **then**
40: $t_{copy} \leftarrow t_{max} - 1$
41: $t_{work} \leftarrow t_{max} - t_{copy}$
42: **return** L

t_{work} for each chunk so the appropriate thread balance can be determined the next time the specific chunk is worked on. To do this, we hold arrays of these variables in memory and update them as-necessary on a per-chunk basis.

D. Application to other Vertex Programs

The approach we've described for label propagation is generally applicable to other iterative vertex-centric graph algorithms. Analytical algorithms like PageRank and K-cores, which have common implementations following a vertex-centric processing model [17], [20], are canonical examples. Considering Algorithm 1, the only changes required would be to replace the contents of vertex state initialization and the processing within Lines 13–19. These chunking methods and optimizations could potentially find wide applicability on KNL architectures, GPUs, and in distributed processing (i.e., reducing communication to only every X processing rounds is an equivalence to our chunking formulation). We note the obvious, however, in that the effects of chunking on solution time and quality will be variable among these different vertex-centric codes and architectures.

IV. Experimental Setup

Our experimental system is the *Bowman* testbed cluster at Sandia National labs. Each node has 96GB DDR and 16GB MCDRAM HBM and a KNL processor with 68 cores. We selected large-scale graph data from a number of sources. The graph properties are listed in Table I. For the directed graphs, we removed their directivity. The diameter shown in the table was determined through an approximate calculation when not explicitly given by the source or given for the directed graph, so should be only considered as a lower bound.

TABLE I
TEST GRAPH CHARACTERISTICS. # VERTICES (n), # EDGES (m), AVERAGE ($davg$) AND MAX ($dmax$) VERTEX DEGREES, AND APPROXIMATE DIAMETER ($\tilde{D}$) ARE LISTED. $B = \times 10^9$, $M = \times 10^6$, $K = \times 10^3$.

Network	n	m	$davg$	$dmax$	$\tilde{D}$
LiveJournal	4.8 M	69 M	18	20 K	18
Friendster	66 M	1.8 B	27	5.2 K	34
Twitter	52 M	2.0 B	37	3.7 M	19
Host	89 M	2.0 B	22	3.4 M	23
uk-2007	105 M	3.3 B	31	975 K	82
wBTER_50	50 M	1.2 B	24	110 K	12
wBTER_100	100 M	2.4 B	24	135 K	12

The Friendster social network graph and LiveJournal graphs were retrieved from the Stanford Large Network Dataset Collection [13]; we include the LiveJournal graph in our tests as it has somewhat become a standard benchmark graph, despite that it could fit entirely in HBM. The Twitter graph is a crawl from the Max Planck Institute [5]. The Host graph is the host-level domain graph from the Web Data Commons 2012 crawl [30]. And the uk-2007 graph is a crawl of the `.uk` domain from the Laboratory for Web Algorithmics [3].

We also generate synthetic graphs using a preliminary version of our wrapped BTER generator [2], which utilizes the BTER graph generator [10] for the scalable generation of graphs with a defined community structure. For these graphs, we set a mixing parameter of $\mu = 0.25$, indicating that 75% of links are contained within communities and 25% of links go across communities. We additionally perform some tests at the smaller scale utilizing the LFR benchmark [11] with a wider range of mixing parameter values. Random graphs like R-MAT or Erdős-Rényi graphs typically don't have any inherent community structure, so we omit them from our testing.

V. Results

Here, we describe the observed experimental performance of our optimizations for graph processing in HBM. Explicitly,

with relation to label propagation in particular, we define the goals of our work to be:

1) **Goal 1:** Demonstrate that HBM work chunking methods do not degrade solution quality
2) **Goal 2:** Demonstrate that HBM work chunking methods can improve time to solution

A. Defining Convergence

In practice, solution convergence for label propagation can be relatively ill-defined. Actual convergence, where no more label updates occur on any given iteration, can occur in a few iterations on networks of a certain structure [21]. However, for the large real-world social and web graphs that we're considering in our experiments, there is generally a long tail of iterations where few updates occur. These updates typically have minimal impact on overall solution quality. We observed while running on the SNAP LiveJournal social graph [12], that label propagation takes over 75 iterations to converge, with an improvement in modularity of only 1% occurring through the last 50 iterations. For our largest test instance, the uk-2007 crawl, convergence can take over 1,000 iterations; we observe that the rate of convergence is loosely dependent on graph diameters and/or mean shortest paths, as one might implicitly infer.

Therefore, often in practice *for large networks*, label propagation is run for either a fixed number of iterations or until some other criteria is reached, such as a modularity amount or the number of labels or the sizes of communities, among others. For the purpose of the following experiments, we define *true convergence* as the case where no label propagations occur. We also consider per-iteration time costs and *relative convergence*, where a chunking method reaches the quality of the Baseline in terms of some ratio of modularity. In general, performance testing for an algorithm like label propagation has the complications of a loosely defined optimization goal and the built-in nondeterminism of the algorithm; we do multiple runs and take the geometric averages when possible.

B. Chunking Parameters Evaluation

As stated, our first goal is to show that introducing work chunking does not have an impact on final solution quality. To experimentally test this, we perform work chunking across a range of *chunks* and *iterations per chunk* on our test graphs. The heatmap given in Figure 1 (left) shows the geometric averaged percentage increase in the number of required iterations across LiveJournal, Friendster, Twitter, Host, and the wBTER graphs over 5 runs. We run either to true convergence or until 100 iterations is reached (we have to omit true convergence here as some graphs – e.g., uk-2007 – take > 1000 iterations and parametric testing was cost-prohibitive). Figure 1 (right) shows the percentage decrease in modularity as a geometric average across all test graphs. For these runs, we show results from the vertex block strategy, as we would consider that the "baseline" partitioning or parallelization approach to be used in practice.

We consider the number of chunks from 2 through 50 with vertex-block chunking, and the iterations per chunk from 1 through 50. Note two things: first, running with a single iteration per chunk is still different from the baseline algorithm, as the processing of vertex order is only randomized within the chunk; and second, the actual iterations performed per chunk is only bounded above by the given number – the chunk might locally converge before i.e. 50 chunk iterations are performed. In these figures *a light color indicates better relative performance*. The range from light to dark in the left plot captures about a $5\times$ increase in iteration cost, while the total range captured in the right plot is approximately 2% modularity difference.

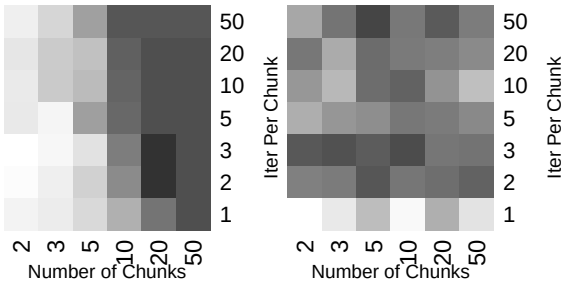

Fig. 1. Heatmaps of the geometric mean number of iterations for convergence relative to the baseline (left) and decrease in modularity (right) relative to the baseline over varying graphs and chunk parameters. A lighter color is better.

We first note the obvious observation, which is that increasing the number of chunks increases the total number of required global iterations, and this increase can be quite considerable – from 20 iterations to over 100 required for Twitter to converge. Additionally, increasing the upper bound on iterations per chunk has a similar affect. These results would be obvious and expected. However, we note that **the solution quality in terms of modularity is minimally affected by chunking**, even across the wide breadth of parameters chosen. While we note that while it takes considerably longer to result in *true convergence* of label propagation, after about 20-40 global iterations (when the iterations per chunk is fewer than 20), the modularity value are usually within a percent of the optimum. Also, its noticed that while running for only a single iteration per chunk on average has slightly better quality, the averaged affect of this is over all chunk counts is well less than a percent. We don't observe any other obvious trends with respect to chunks per iteration. We also calculated the rand index versus the baseline output for each test and observed the same result – we omit the corresponding heatmap for brevity.

We also observed the surprising effect that modularity appears to increase with increasing chunks on a number of graphs, even up by several percent. On average, the modularity increases with increasing chunks on LiveJournal, Host, and the wBTER graphs. We explain this relative quality gain due to the fact that breaking up the graphs into vertex block chunks is possibly resulting in better refinement within the chunk boundary – e.g. smaller communities contained entirely within a chunk are more often labeled correctly or less likely

consumed by a larger community's label. We justify this explanation with the fact that this effect is only noticeable with block and explicit partitioning for work chunking and not with random assignment. Therefore, for the block methods, this effect would correspondingly only be demonstrable with a well-ordered initial graph. We additionally studied the number of iterations per convergence for the other three partitioning scenarios (block edge, PuLP, random), and noted that the block edge and PuLP both converge at approximately the same rate as vertex block. The random assignment of vertex-to-chunk results in overall poor convergence performance in terms on number of iterations. This effect is likely related to the effect on quality mentioned above.

Overall, we can state that these results demonstrate chunking will eventually converge to a solution comparable to the baseline approach, albeit by taking a greater number of iterations.

1) Running on LFR Benchmark: We additionally evaluate the impact of chunking on the LFR benchmark [11] (parameters $n = 10,000$, $k = 15$, $maxk = 500$, $t1 = 2$, $t = 1$), with the mixing parameter $\mu = 0.05 \ldots 0.6$. Larger mixing parameters $\mu \geq 0.7$ were noted to result in convergence of baseline and chunked label propagation to a single community. Figure 2, gives the relative number of iterations for convergence relative to the baseline (left) and the normalized mutual information (NMI) of the Baseline (*Base*) versus chunking with 5 chunks and 5 iterations per chunk (*Chunk_5_5*) and 50 chunks and 50 iterations per chunk (*Chunk_50_50*). The number of global iterations required for convergence is extremely variable – from 5 iterations for baseline on the $\mu = 0.05$ test graph to over 500 iterations with 50 chunks and 50 per-chunk iterations on the $\mu = 0.6$ test graph.

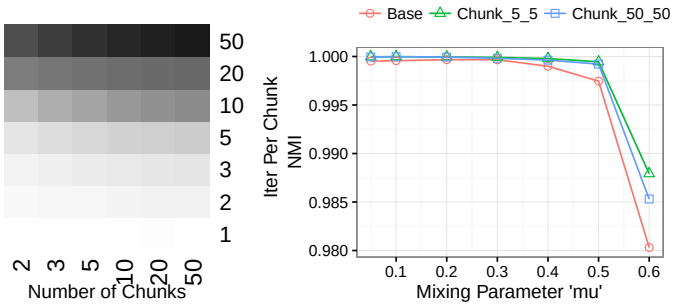

Fig. 2. Heatmaps of the number of iterations (geometric mean over varying μ) for convergence relative to the baseline (left) and LFR NMI plot (right) for varying chunk parameters.

As before, we observe the dependence of number of global iterations required on chunking for these tests. We note that the solution quality relative to baseline actually slightly *increases* with chunking in these tests as before. Regardless, based off of the preceding experimental observations, we state in regards to **Goal 1** that while it may increase the required number of iterations, **chunking does not have a large impact on the solution quality of label propagation** relative to the baseline in terms of modularity. The next question we seek to answer, is whether we can optimize for HBM such that the increased cost in number of iterations is offset by a decrease in per-iteration execution time.

C. Optimizing Time to Convergence

Here, we'll discuss the relative per-iteration speedups achieved for each of the considered optimizations for work chunking in HBM. As we've observed, a greater number of chunks will increase the total global iteration cost. However, since the total iterations required for true convergence of the baseline on a number of instances is unfeasibly too large to utilize in a full parametric study, we restrict these test instances to a maximum of 40 total global iterations. The above results indicated that a minimum number of chunks possible should be used for fastest solution convergence, so we use the minimal number of chunks possible for testing. This minimum number depends on the partitioning strategy as discussed below; however, it is in general about 5 – with an expection that when we overlap communication, the required chunks doubles. We run LiveJournal with 5 chunks for comparison although it can fit entirely in HBM.

1) Hybrid Mode: The first method we investigated to improve solution time was through the use of *Hybrid* mode, or 50/50 split Cache mode/Flat MCDRAM model. We considered two variants. The first variant. **Hybrid 1**, only allocates fixed thread-owned data into HBM. Since this data is static between tests, it eliminates explicit transfer costs and effectively offloads those costs from the implicit caching in the other 50% of the HBM space. **Hybrid 2** has an opposite allocation scheme, where algorithmic data is allocated into HBM and the thread-owned structures are handled by caching. In practice, we found Hybrid Mode 2 to have a very big overall slowdown among all tests (approximately $3\times$ on average versus Baseline). It is likely due to the additionally limited HBM space and necessary requirements to increase the number of chunks that contributed to this slowdown. Additionally, since the graph structure and per-vertex memory accesses has some degree of locality associated with them versus the thread-owned label counts hash tables, the predictive caching might be relatively more effective in handling these data structures.

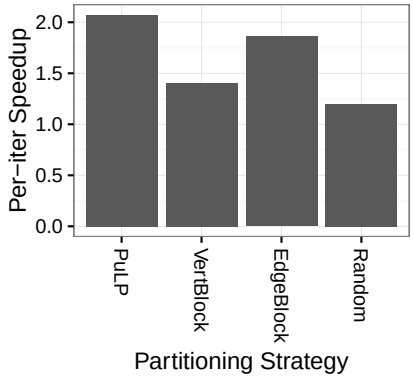

Fig. 3. Speedups achieved by using *Hybrid* mode.

We show in Figure 3 the speedups for **Hybrid 1** versus the Baseline running in *Cache* mode. We note a relative, and sometimes significant in the case of the Twitter graph, improvement when running on **Hybrid 1**. Future work here might investigate a hybrid scheme, where perhaps just the labels themselves or the the labels and thread-owned data are held in HBM and the graph data is handled by implicit caching. Overall, running in *Hybrid* mode demonstrates some degree of speedup for all instances relative to Baseline in *Cache* mode.

2) Partitioning Methodology: The partitioning method for chunk creation in particular can have a drastic impact on memory allocation cost. Consider the Twitter graph, which has an extremely skewed degree distribution. With even vertex block partitioning it requires 200 chunks such that all adjacent edges to vertices in the most heavyweight chunk can be stored in HBM along with all the necessary algorithmic data. With edge block partitioning or even random partitioning, the number of chunks required drops to only 5. The partitioning methodology also has an impact on per-iteration timing. In Figure 4 we plot the geometric mean speedups versus the baseline for running 40 iterations across all graphs under various partitioning methodologies and chunking parameters. We additionally plot the geometric mean of the improvement relative to modularity.

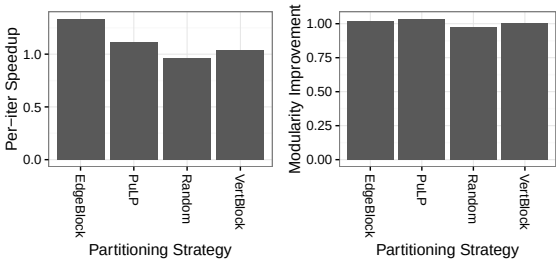

Fig. 4. Speedups and Modularity impact of partitioning methods.

From Figure 4, we observe that explicit partitioning has the greatest impact on both per-iteration cost and relative modularity gain. The block partitioning strategies improve on average speed up per-iteration time relative to the baseline with negligible impact on solution quality. Random assignment of vertices to blocks, however, degrades solution quality and shows negligible speed improvements for chunking versus the baseline. These impacts again should be noted in the context that the graphs as-is are likely relatively well-ordered. We also note that the explicit gain in modularity comes at the obvious cost of computing the partitioning itself. The modularity differences shown here, however, are relatively minor, only about 3% in each direction.

3) Overlapping Communication: We note that one of the largest costs of chunking is associated with the explicit transfer of the graph from DDR to HBM. As we've described, we implemented a method using OpenMP sections that effectively will overlap our communication and computation portions. This enables us to effectively hide this transfer cost, often only by sacrificing a single work thread in order to do the transferring. As we've described previously, we dynamically adjust the number of transfer threads as-needed based on prior work/transfer time imbalances; this imbalance is dependent on the number of chunks and number of iterations per chunk. In Figure 5 we plot the impact of communication overlapping on the several test graphs.

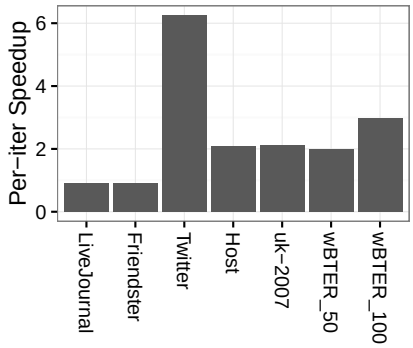

Fig. 5. Per-iteration relative speedup benefit of overlapping communication for each graph averaged over all partitioning strategies.

The figure gives the relative speedup with overlapping communication averaged across all partitioning methods and for all graphs when running with 5 iterations per chunk. The results in Figure 5 are surprising, but consistent. We observe that the impact of overlapping communication can be quite significant, and correlates with the skew on each graph. On average, about a $2\times$ speedup is observed for most graphs, which could be expected. For graphs with larger speedups, we note that this results from the required increase in chunks, which can result in more consistent work/transfer balance during overlapping communication and can also considerably decrease per-iteration work time as a result of improved cache performance.

D. Discussion: Time to Convergence

In Figure 6 we show the "optimal" per-iteration speedup measured for each test graph for 40 iterations of label propagation. The level of speedup that work chunking achieves is quite variable and appears to correlate with the degree skew of the graph. For the Twitter graph, we observe considerable speedup improvement with edge block partitioning and overlapping work with transfer costs. Even in the worst test case, the Friendster graph, we still observe about 10% reduction in per-iteration cost. Regardless, the original question we sought for **Goal 2** was whether or not chunking can improve time to solution, not just time per-iteration. Our answer: **it depends**.

As part of this discussion we take our two most extreme graph instances, Twitter and Friendster. We define them that way in that Twitter convergences to *true convergence* very quickly and benefits greatly from our optimizations, while Friendster convergences relatively slowly compared to the other instances and benefits least from our optimizations. We plot in Figure 7 the convergence of each (Friendster on top, Twitter on bottom) relative to modularity with the Baseline and

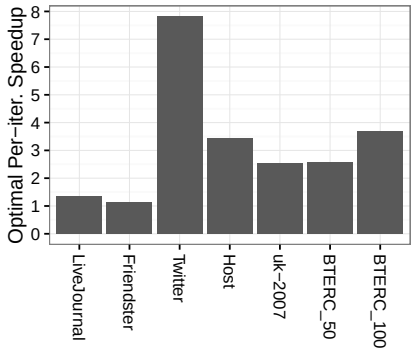

Fig. 6. Optimal per-iteration speedups measured with selection of partitioning strategy and overlapping of communication with computation.

various chunking parameters. We show convergence versus global number of iterations (left) and total execution time (right). We run Twitter to true convergence and fix Friendster for 40 iterations.

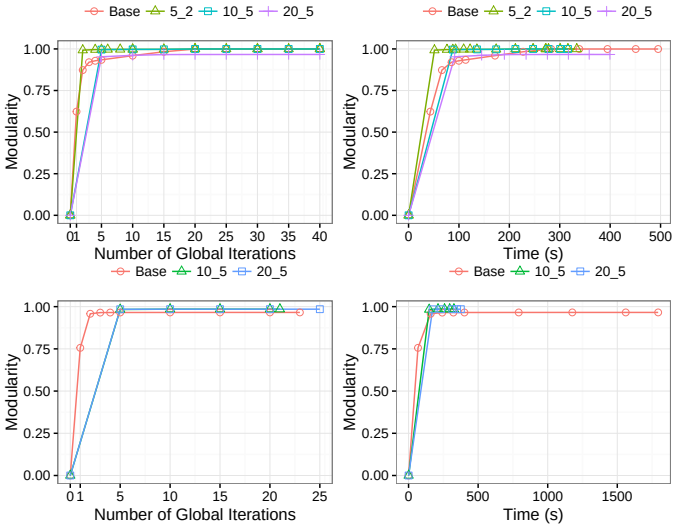

Fig. 7. Modularity vs. global iteration counts (left) and timing (right) for a variety of chunking parameters on Friendster (top) and Twitter (bottom). Labels can be read as $C_{num}\_C_{iter}$

We note in Figure 7 the previously observed statement that a majority of modularity gain occurs in about the first five iterations [21] of the algorithm. And as especially the case in real-world graphs, there can be a long tail of minor updates which don't have a major impact on the total solution quality. Effective utilization of HBM results in Twitter converging to true convergence more quickly than the baseline completes five iteration in *Cache* mode. On Friendster, the rate of modularity gain (as defined by the modularity after 5 iterations) appears to be higher than the Baseline on all of the chunking variants, though this is with an edge block partitioning and the graph is likely well-ordered. We note on both instances that final modularity reached is approximately equal.

In reference to Goal 2: Work chunking for label propagation in general appears to improve time to solution for all of our test instances when we assume a loose convergence criteria. With a fixed criteria of *true convergence* for label propagation, then the benefits of HBM work chunking is dependent on the graph. In our observations, skewed small-world graphs which take the longest time to process in *Cache* mode offer the greatest room for improvement in *Flat* mode work chunking. When overlapping communication, it is possible to run 5 or fewer iterations, which will minimally impact final solution quality and best utilize HBM. Therefore, convergence to a looser criteria such as some minimum percent modularity gain or some minimum number of updates is likely to occur more quickly with work chunking.

E. Discussion: Chunking Generalization

We also investigate the generalization of our approaches towards other iterative vertex-centric programs, specifically k-cores and PageRank. We run K-cores to true convergence, where every vertex gets assigned to it its true coreness value. We run PageRank for a fixed 100 iterations. We note that chunking for k-cores will greatly increase the number of required global iterations (e.g., by approximately 3× on Twitter with 5 local iterations per chunk) for convergence, as we've seen with true convergence with label propagation. As a result, end-to-end times generally increase. However, we do observe some per-iteration speedup when overlapping communication and performing only a single local iteration per chunk (possibly a result of cache pre-heating), at no cost to the number of global iterations. This speedup, however, is relatively modest and on average well less than 25%. We observe a similar magnitude of speedup when chunking and overlapping communication for PageRank.

While we note that the speedups observed in these general instances are less than what was observed with label propagation, we still note that the techniques explored in this paper have some capability to generalize. It is likely that the greater speedups we observed were a result of the hash table performance being significantly slower in the cache mode test instances. Our observations can also be further generalized to distributed environments with a partitioned graph, where the equivalence to chunking would be communication after only every *nth* local iteration. As we've observed minimal impact on solution quality, distributed implementations of label propagation and its variants might be able to drastically reduce synchronization and communication requirements when implemented in a bulk synchronous processing model.

VI. Conclusions

Our primary observations from the study of work chunking on the Intel Knights Landing High Bandwidth Memory are

1) Work chunking has minimal observed impact on final solution quality.
2) Increasing the number of chunks and number of iterations per chunk increases the number of global iteration for full convergence, with the latter having a slightly larger effect.
3) However, work chunking with explicit HBM usage can be used to improve time to solution relative to a baseline algorithm running in cache mode.

4) Overlapping communication with work is critical for optimal performance.
5) Balancing the size of chunks in terms of memory footprint is another key performance consideration.
6) Explicit partitioning shows minor performance benefit relative to block strategies – though this assumes a well-ordered initial graph. With a randomly ordered graph, explicit partitioning would offer a much greater benefit, though obviously at a cost of computing the partition.

The are many avenues for future work. Further refinement of the optimal chunk size and partitioning strategy for a given graph class might be promising future work. Multi-tiered chunking, in which a few key critical (perhaps in terms of some centrality measure) vertices remain resident in HBM and are updated every iteration might enable faster convergence. Applying and exploring the optimizations described within this work across a number of different algorithms and analytical domains might provide for promising future results.

Acknowledgements: We thank the ASC Advanced Architectures test-bed team at Sandia National Laboratories for supplying and supporting the systems used in this paper. Additionally, we thank Jonathan Berry, Cynthia Phillips, and Mehmet Deveci for helpful discussions during the development of this work. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA-0003525. The research presented in this paper was funded through the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories, in the context of the Multi-Level Memory Algorithmics for Large, Sparse Problems Project.

REFERENCES

[1] M. A. Bender, J. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues, "Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 835–846.

[2] J. Berry, C. Phillips, S. Rajamanickam, and G. M. Slota, "Scalable community detection benchmark generation," in *Abstract for the SIAM Combinatorial Scientific Computing Conference (CSC)*, 2018.

[3] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.

[4] C. Cantalupo, V. Venkatesan, and J. R. Hammond, "User extensible heap manager for heterogeneous memory platforms and mixed memory policies," 2015.

[5] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2010.

[6] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

[7] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[8] C. Gaiteri, M. Chen, B. Szymanski, K. Kuzmin, J. Xie, C. Lee, T. Blanche, E. C. Neto, S.-C. Huang, T. Grabowski, T. Madhyastha, and V. Komashko, "Identifying robust communities and multi-community nodes by combining top-down and bottom-up approaches to clustering," *Scientific reports*, vol. 5, 2015.

[9] S. Gregory, "Finding overlapping communities in networks by label propagation," *New Journal of Physics*, vol. 12, no. 10, p. 103018, 2010.

[10] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C424–C452, 2014.

[11] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, no. 4, pp. 1–5, Oct. 2008. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.78.046110

[12] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[13] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[14] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[15] X. Liu and T. Murata, "Advanced modularity-specialized label propagation algorithm for detecting communities in networks," *Physica A: Statistical Mechanics and its Applications*, vol. 389, no. 7, pp. 1493–1500, 2010.

[16] 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.

[17] 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.

[18] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys*, 2015.

[19] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[20] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2013.

[21] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

[22] A. L. Schmidt, F. Zollo, M. Del Vicario, A. Bessi, A. Scala, G. Caldarelli, H. E. Stanley, and W. Quattrociocchi, "Anatomy of news consumption on facebook," *Proceedings of the National Academy of Sciences*, p. 201617052, 2017.

[23] B. Shao, H. Wang, and Y. Xiao, "Managing and mining large graphs: systems and implementations," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 589–592.

[24] G. M. Slota, S. Rajamanickam, , and K. Madduri, "Order or shuffle: Empirically evaluating vertex order impact on parallel graph computations," in *Graph Algorithms Building Blocks Workshop (GABB)*, 2017.

[25] G. M. Slota, S. Rajamanickam, and K. Madduri, "Complex network partitioning using label propagation," *SIAM Journal on Scientific Computing (SISC)*, vol. 38, no. 5, pp. S620–S645, 2016.

[26] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.

[27] A. Sodani, "Knights landing (knl): 2nd generation intel® xeon phi processor," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 2015, pp. 1–24.

[28] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[29] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 2013, pp. 507–516.

[30] "Web data commons," http://webdatacommons.org/.

[31] J. Xie, M. Chen, and B. K. Szymanski, "Labelrankt: Incremental community detection in dynamic networks via label propagation," in *Proceedings of the Workshop on Dynamic Networks Management and Mining*. ACM, 2013, pp. 25–32.

[32] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 140–149.

[33] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," 2002.