Approximate Computing Techniques for Iterative Graph Algorithms

Ajay Panyala[1], Omer Subasi[1], Mahantesh Halappanavar[1], Ananth Kalyanaraman[2], Daniel Chavarría-Miranda[3], and Sriram Krishnamoorthy[1]

E-mail: {FirstName.LastName}@pnnl.gov, ananth@eecs.wsu.edu, daniel@trovares.com
[1]Pacific Northwest National Laboratory [2]Washington State University [3]Trovares Inc.

***Abstract*—Approximate computing enables processing of large-scale graphs by trading off quality for performance. Approximate computing techniques have become critical not only due to the emergence of parallel architectures but also due to the availability of large scale datasets enabling data-driven discovery. Using two prototypical graph algorithms, PageRank and community detection, we present several approximate computing heuristics to scale the performance with minimal loss of accuracy. We present several heuristics including loop perforation, data caching, incomplete graph coloring and synchronization, and evaluate their efficiency. We demonstrate performance improvements of up to 83% for PageRank and up to $450\times$ for community detection, with low impact on accuracy for both the algorithms. We expect the proposed approximate techniques will enable scalable graph analytics on data of importance to several applications in science and their subsequent adoption to scale similar graph algorithms.**

I. Introduction

Approximate computing methods trade off quality for performance. Approximate computing covers a wide range of techniques based on modifications in algorithms, runtime systems, compilers, and hardware [1]. The need for approximate computing is driven from the need to efficiently process large-scale data emerging from all aspects of science and technology, as well as the fundamental changes in computer architecture with stringent restrictions on power consumption and massive parallelism.

In this paper, we present multiple approximate computing techniques as applied to graph algorithms that have an iterative structure to them, where in each iteration all vertices (alternatively, edges) of the graph are processed. This is a common feature in many graph algorithms that fit the vertex-centric programming model [2]. We propose multiple generic approximate computing techniques (Section II) aimed at significantly improving performance with minimal impact on output quality. Key techniques include *loop perforation*, *vertex/edge ordering*, and *threshold scaling*.

For the purpose of implementation and testing, we consider two prototypical iterative graph algorithms—one for *PageRank* (Section III) and another for *community detection* (Section IV)—and show how the proposed techniques manifest themselves in practice. Our work is motivated from the emerging importance of graph algorithms for analytics workloads with an increasing need to perform analytics on dynamic data of unwieldy sizes, where approximate computing often becomes imperative.

Given a directed graph $G = (V, E)$, PageRank of a vertex notionally represents its relative importance in the graph [3]. PageRank is an important centrality metric with numerous applications. We present approximate computing techniques for the PageRank algorithm by focusing on the notion of importance of the outgoing edges and selectively updating only a subset of the neighborhood for a given vertex. Further, we preorder the neighbor lists based on the outdegree of the neighbors, which we use as a proxy for relative importance. Preordering the neighbor lists enables us to amortize costs for memory access as well as quickly identify cutoffs for computation. We detail our method in Section III. Empirically, we observe substantial improvement in the convergence of PageRank, which we detail in Section V. We compare the performance and correctness of our approximate algorithm with the state-of-the-art STIC-D [4] algorithm.

Given an undirected graph $G(V, E, \omega)$, community detection aims to compute a partitioning of V into a set of tightly-knit communities (or clusters). Community detection has emerged as one of the most frequently used graph structure discovery tools in a diverse set of applications [5]. We employ several approximate computing techniques to improve the performance of an agglomerative technique based on modularity optimization [6], one of the fastest methods for community detection albeit with several limitations. As described in Sections II and IV, our techniques take advantage of some key properties of the underlying algorithm in order to save on computation work and memory lookups. We observe significant improvements in performance (Section V) from these techniques. We compare the performance and correctness of our approximate computing techniques using the state-of-the-art Grappolo [7] algorithm.

In summary, we make the following contributions in this work:

- Design of multiple generic approximate computing techniques suitable for iterative graph algorithms;
- Analysis of the memory access characteristics of the PageRank algorithm to identify and isolate the most expensive memory accesses;
- Design of approximate computing algorithms that minimize very expensive, latency-inducing irregular memory accesses;
- Empirical validation of the correctness of the approximation in terms of multiple metrics;

0-7695-6326-0/17/$31.00 ©2017 IEEE
DOI 10.1109/HiPC.2017.00013

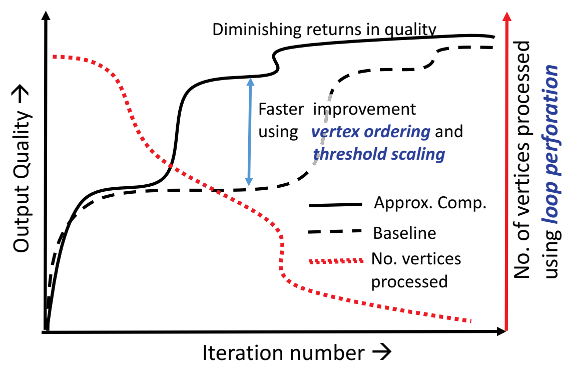

Diminishing returns in quality
Faster improvement using vertex ordering and threshold scaling
Approx. Comp.
Baseline
No. vertices processed
Output Quality →
Iteration number →
No. of vertices processed using loop perforation

Figure 1. A schematic illustration of our main approximate computing ideas applied to iterative graph algorithms. The steep climbs in quality are a typical feature in iterative algorithms that perform coarsening—such as [7].

- Extensive evaluation demonstrating performance improvements of up to 83% for PageRank and up to $450\times$ for community detection; and
- Evaluation and comparison against a state-of-the-art PageRank approximation algorithm (STIC-D).

II. Approximate Computing Techniques

For many applications, the overall computational cost can be significantly reduced if an exact solution is not required. Under approximate computing, portions of a computation are skipped or transferred to more efficient but imprecise versions such that the overall quality of the solutions is not significantly perturbed. Approximate computing has the potential to improve performance and energy efficiency at the expense of solution accuracy. Below, we discuss the approximate computing techniques we employ in the context of iterative graph algorithms. Figure 1 illustrates the key ideas.

A. Loop perforation

Broadly, loop perforation is the idea of randomly or selectively skipping a portion of the operations for a given iteration of a loop. There are different ways to implement this technique, taking advantage of appropriate properties of the underlying iterative algorithm. But the main idea is to effect a progressive decrease in the work performed within each iteration, as the algorithm proceeds, as shown in Figure 1. We interpret loop perforation in two similar ways to match the needs of the graph algorithm under consideration.

For PageRank, the algorithm arranges the neighbors of each vertex in a non-increasing order of importance, and progressively skips operation on the neighbors as the algorithm proceeds. We use the indegree of a neighbor as a proxy for importance, which can be easily replaced with other measures of importance. The algorithm is described in Section III. The impact on performance is provided in Section V.

We interpret loop perforation for community detection through the idea of *early termination* of the computation, as the algorithm progresses, for a selected subset of vertices at each iteration. The key property we exploit is one of *diminishing returns* exhibited by the underlying algorithm. To illustrate this, let us consider the community detection algorithm in [7]. This algorithm takes an iterative approach, where every vertex is processed at every iteration, and at the end of each iteration a convergence criterion is evaluated: is the net quality gain (measured by the modularity measure [8]) achieved by the current iteration above a certain positive threshold θ? If so, the algorithm proceeds to the next iteration; otherwise it terminates the current sequence of iterations (aka "phase").

Because each step in the algorithm attempts to improve modularity, the overall quality of the solution is guaranteed to be non-decreasing as the algorithm progresses through its iterations (as shown by the quality curves in Figure 1). Interestingly, we observed in [7] that the quality increases at a faster rate in the initial iterations, while in the later iterations it effectively plateaus (as also illustrated in Figure 1). We observed this pattern for every input we tested in [7]. In fact, the reason for this behavior is that vertices tend to migrate from one community to another more frequently during the initial iterations, but as the algorithm progresses, they settle in.

This observation leads us to the idea of *early termination*, where we detect vertices that are less likely to change their state (i.e., migrate) in future iterations and retire such vertices from consideration in subsequent iterations. There are many ways to implement this simple idea, and Section IV describes one possible mechanism.

B. Vertex/Edge ordering

Vertex ordering represents the order in which the vertices are considered for execution during a given iteration. It has been well-studied that vertex ordering has a tremendous impact on performance of graph algorithms [9], [10]. Vertex ordering becomes a considerable challenge in the context of parallel execution where ordering becomes dependent on several parameters including the runtime system. We, therefore, employ vertex ordering as an approximate computing technique in this work.

Vertex ordering in community detection is implemented through graph coloring [7], [11]. Graph coloring assigns a color (identified by integer ids) to each vertex such that no two vertices connected by an edge receive the same color [12]. The latter condition makes it a *distance*-1 coloring problem. We employ two variants of this coloring problem, *complete* and *incomplete*. In the complete scheme, every vertex receives a valid color (consistent with the distance-1 condition). In the incomplete scheme, however, only a subset of vertices receives color consistent with the distance-1 criterion, while the remaining are bundled up into a single arbitrary color.

Subsequently, within every iteration, we go through each color class in an arbitrary order and process all vertices belonging to that color class in parallel—with the goal of not processing any two neighbors concurrently. This strategy ensures that more parallelism is available for the larger color classes. Consequently, both the performance (runtime per iteration, convergence rate) and quality of the output depend on the schemes used, which we evaluate in Section V. Figure 1 shows the targeted behavior, where the partial ordering produced by coloring improves performance without affecting quality.

Similarly, for PageRank, we combine vertex ordering with loop perforation by ordering the outgoing neighbors of each vertex based on the neighbor's indegree. The indegree of a neighbor represents its importance. By progressively skipping operations on the outgoing neighbors, the set of vertices processed changes during each iteration. Further, this ordering provides us a mechanism to determine in near-constant time the edges along which operations need to be performed.

C. Threshold Scaling

We use another approximate computing technique, *threshold scaling*, for iterative graph algorithms. Threshold scaling is used effectively in many algorithms, such as the auction algorithm for bipartite weighted matching [13]. We empirically evaluate its impact on performance for the community detection algorithm where there is a graph coarsening step. The basic idea for threshold scaling is to make the threshold for convergence more stringent during the later stages of the algorithm, so that the algorithm iterates longer in those stages, albeit on a smaller graph. On the other hand, the less stringent setting in the early stages of the algorithm allows the coarsening to happen sooner than later, thereby potentially improving performance. This targeted behavior is illustrated in Figure 1. We explain this technique in the context of the community detection operation in Section IV.

In addition to these three approximate computing techniques, we also considered a few related techniques, such as synchronization and data caching, in our study. However, we skip them from a detailed treatment in this paper.

III. Approximation Techniques for PageRank

Google's PageRank [3] is a method for computing the ranking of webpages in their search engine results. Usually, PageRank evaluates a large set of web pages connected through hyperlinks. PageRank [3] implements the "random surfer model," where a web surfer walks the outgoing links of an initial vertex, chosen uniformly at random with probability d. At random or when the surfer ends up at a page with no outgoing links, the surfer, with probability $(1 - d)$, jumps to an entirely new page and restarts the random walks from this new page. The output of PageRank is a score for each page on the web that determines its importance. The web usually can be represented as a directed graph whose vertices are HTML pages and edges hyperlinks. Note that PageRank applies to almost any directed graph, not just web graphs. For example, PageRank is used for solving problems in various domains such as chemistry, bioinformatics, bibliometrics, and social networks analysis [14].

Consider a directed graph G = (V, E) with vertex set V and edge set E. For a given vertex V_i, let $\mathrm{IN}(V_i)$ be the set of incoming vertices. The PageRank score for vertex V_i is defined [3] by the equation:

$$\mathrm{PR}(V_i) = \frac{1-d}{|V|} + d \times \sum_{j \in \mathrm{IN}(V_i)} \mathrm{contribution}(V_j) \qquad (1)$$

The damping factor, d, usually set to 0.85 [3], denotes the probability of jumping from a given vertex to another random vertex in the graph. We start by initializing all vertices to an initial PageRank score of $\frac{1}{|V|}$. The PageRank computation iteratively computes the rank value of each vertex (using Equation 1) until convergence.

There is a notion of the PageRank value for the entire graph (**global PageRank**), which corresponds to the summation of the PageRanks of its vertices: $\mathrm{PR}(G) = \sum_i \mathrm{PR}(V_i)$. When comparing different algorithms to compute PageRank, we can compare their accuracy by: (*i*) comparing the global PageRanks they compute for the same datasets, and (*ii*) comparing the relative ordering imposed on the vertices by the resulting vertex PageRanks (as well as comparing the PageRank values of individual vertices).

Algorithm 1 shows an overview of the code for our baseline (exact) implementation. On each iteration, we first traverse the outgoing edges for each vertex V_i and store V_i's contribution to each of its neighboring vertices U_j in a contribution list (of size $|E|$). *OffsetList* (constructed during the preprocessing stage) gives the index into the *ContributionList* for each U_j where V_i's contribution to U_j is stored. Then, the code traverses the incoming edges for each vertex V_i and gets V_i's contributions from each vertex X_j along the incoming edges. The contributions from all incoming vertices are summed over, and the PageRank value for V_i is computed using Equation 1. The algorithm computes the maximum absolute difference between the previous PageRank and the new PageRank values for all vertices. The PageRank array is updated to the newly computed PageRank. The process is repeated until global difference is less than the specified threshold. We perform the incoming and outgoing edge traversals in parallel and perform the PageRank computation in double-precision floating point for greater accuracy. We used Intel Threading Building Blocks' `parallel_for` construct to parallelize our computation.

Loop Perforation: We observe that, in general, the change to the PageRank values of individual vertices decreases in magnitude over time. In turn, this results in only minimal changes in the contribution such a vertex makes to its outgoing neighbors. This is especially true for vertices with high outdegree because a vertex's contribution to its outgoing neighbors is its PageRank value divided by its outdegree.

We exploit this characteristic to *freeze* the contributions of a source vertex V_i along outgoing edges leading to target

Algorithm 1 Pseudo-code for our baseline (exact) PageRank implementation

```
PageRank($0 \dots |V|$) ← $\frac{1}{|V|}$
globalDiff ← $\infty$
while globalDiff $> 10^{-6}$ do
    maxDiff = $-\infty$
    for each $V_i = 0 \dots |V|$ do                              ▷ Parallel loop
        contribution ← PageRank($V_i$) / OutDegree($V_i$)
        for each $U_j \in$ OutList($V_i$) do
            ContributionList(OffsetList($U_j$)) ← contribution

    for each $V_i = 0 \dots |V|$ do                              ▷ Parallel loop
        sum ← 0
        for each $X_j \in$ InList($V_i$) do
            sum ← sum + ContributionList($X_j$)
        PreviousPageRank($V_i$) ← PageRank($V_i$)
        PageRank($V_i$) ← $(\frac{1-d}{|V|}) + d * sum$

        diffRanks ← |PageRank($V_i$) - PreviousPageRank($V_i$)|
        maxDiff ← max(diffRanks, maxDiff);                  ▷ Parallel Reduction
    globalDiff ← maxDiff
```

vertices. The idea is that the updates of a source vertex V_i's contribution is only offered to the vertices along its outgoing edges when it will have a bigger impact on the overall PageRank results. Hence, we do not update the target vertices' contributions when V_i's contribution is small, instead using the previous (frozen) value of the contribution.

In addition to reducing the number of memory write operations, the freezing strategy reduces the number of irregular memory accesses. However, checking every access for potential freezing can still be expensive. Therefore, we design a strategy to minimize the number of operations required to determine updates that are candidates for freezing. The optimized algorithm in shown in Algorithm 2.

We sort the outgoing edge list of each vertex V_i based on their target vertices indegree during the graph building and pre-processing stage. The sorted list (for each V_i) is organized into *bins* based on indegree. Our optimized implementation of PageRank uses a fifth vector named *BinIndexList* that stores the bins for all vertices in V. *BinIndexList*(V_i) is a subset of *BinIndexList* that represents all bins for a vertex V_i. Bin x in *BinIndexList*(V_i) corresponds to the index position in *OffsetList* for V_i's outgoing edges whose target vertex indegree is between 2^{x-1} to 2^x. The index position determines the outgoing edges to be frozen.

All the vertices (neighbors of V_i) from the first bin until bin x can be frozen. This change would be reflected in our algorithm where vertex V_i contributes only to vertices (in its outgoing vertex list) which start from the index position given by bin x. Since *BinIndexList* is built during the pre-processing stage, the starting index position is later retrieved during the PageRank computation using

$$\frac{\log_2(\frac{10^{-11}}{(contribution - prev\_contribution)})}{2}$$

as shown in Algorithm 2. Thus, we effectively traverse only a subset of V_i's outgoing edges. By doing so, we significantly save on memory accesses to *ContributionList* and *OffsetList*. The memory writes to *ContributionList* are not contiguous. Hence, by reducing these irregular memory accesses, we gain significant performance improvement.

The maximum number of bins needed for each vertex is $\lceil(\log_2(\mathit{MaxInDegree}(G))\rceil$. We call this the maximum bin count value bc required for each vertex. Therefore, the total size required for bins of all vertices is $bc|V|$. bc ranges from 10 through 27 for the datasets used in this paper, depending on the graph's *MaxInDegree* value.

Algorithm 2 Pseudo-code for our approximate PageRank algorithm

```
PageRank($0 \dots |V|$) ← $\frac{1}{|V|}$
globalDiff ← $\infty$
while globalDiff $> 10^{-6}$ do
    maxDiff = $-\infty$
    for each $V_i = 0 \dots |V|$ do                              ▷ Parallel loop
        contribution($V_i$) ← PageRank($V_i$) / OutDegree($V_i$)
        prevContribution($V_i$) ← PreviousPageRank($V_i$) / OutDegree($V_i$)

        BinNo ← $log2(\frac{10^{-11}}{(contribution - prevContribution)})/2$

        OutListIndex ← BinIndexList[$V_i$*bc + BinNo]
        for each $U_j$ = OutListIndex... |OutList($V_i$)| do
            ContributionList(OffsetList($U_j$)) ← contribution($V_i$)

    ...
    Remaining code same as shown in Algorithm 1
    ...
```

Analysis of Memory Overhead: The data structures used are: *ContributionList* (size $|E|$), *OffsetList*(size $|E|$), and *InList* and *OutList* each of size $|V|$ that store the bounds for the incoming and outgoing edge lists for each vertex. *PageRank* and *PreviousPageRank* vectors (each of size $|V|$). Therefore, the total memory required for the baseline algorithm is $4|V| + 2|E|$. The approximate PageRank algorithm uses an additional *BinIndexList* vector of size $bc|V|$, leading to a total memory footprint of $4|V| + 2|E| + bc|V|$.

IV. Approximate Computing for Community Detection

Given an undirected graph $G(V, E, \omega)$, where V is the set of vertices, E is the set of edges and $\omega(.)$ is a weight function that maps every edge in E to a non-zero, positive weight. We use n and m to denote the number of vertices and the sum of the weights of all edges in E respectively. We denote the neighbor list for vertex i by $\Gamma(i)$. A *community* within graph G represents a subset of V.

In general terms, the goal of *community detection* is to partition the vertex set V into a set of tightly knit (non-empty) communities—i.e., the strength of intra-community edges within each community significantly outweighs the strength of the inter-community edges linked to that community. Neither the number of output communities nor their size distribution is known *a priori*.

Let $P = \{C_1, C_2, \dots, C_k\}$ denote a set of output communities in G, where $1 \leq k \leq n$, and let the community containing vertex i be denoted by $C(i)$. Then, the goodness of such a community-wise partitioning P is measured using the *modularity* metric, Q, as follows [8]:

$$Q = \frac{1}{2m} \sum_{i \in V} e_{i \to C(i)} - \sum_{C \in P} \left(\frac{a_C}{2m} \cdot \frac{a_C}{2m}\right), \quad (2)$$

where $e_{i \to C(i)}$ is the sum of the weights of all edges connecting vertex i to its community, and a_C is the sum of the weights of all edges incident on community C. The problem of community detection is then reduced to the problem of modularity maximization, which is NP-Complete [15].

The Louvain algorithm proposed by Blondel *et al.* [6] is a widely-used efficient heuristic for community detection. *Grappolo* was recently developed as a parallel variant of the Louvain algorithm by Lu *et al.* [7]. We build on Grappolo for this work and implement different approximate computing techniques. In this section, we focus on the core ideas behind incorporation of these techniques into the Grappolo algorithm; the reader is referred to [7] for more details about the Grappolo algorithm.

Grappolo is a multi-phase multi-iteration algorithm, where each phase executes multiple iterations as detailed in Algorithm 3. Within each iteration, vertices are considered in parallel (Line 8) and decisions are made using information from the previous iteration, and thus, eliminating the need for explicit synchronization of threads. If coloring is enabled, then vertices are partitioned using the color classes (Line 2). The threads synchronize after processing all the vertices of a color class (within the for loop on Line 6), and therefore, use partial information from the current iteration. The algorithm iterates until the modularity gain between successive iterations is above a given threshold θ (Lines 17-20).

Within each iteration, the algorithm visits all vertices in V and makes a decision—whether to change its community assignment or not. This is achieved by computing a modularity gain function ($\Delta Q_{i \to t}$), by considering the scenario of vertex i potentially migrating to each of its neighboring communities (including its current community) (t), and selecting the assignment that maximizes the gain (Lines 10-13).

At the end of each phase, the graph is coarsened by representing all the vertices in a community as a new level "vertex" in the new graph. Edges are added, either as self-edges (an edge from a vertex to itself) with a weight representing the strength of all intra-community edges for that community, or between two vertices with a weight representing the strength of all edges between those two communities. The algorithm iterates until there is no further gain in modularity achieved by coarsening (Lines 17-20).

Loop Perforation via Early Termination: For community detection, we implement the loop perforation technique by a scheme that detects and terminates vertices (from being considered for processing) as the algorithm progresses through its iterations. We refer to this technique as *early termination*. This idea was introduced in Section II. In

Algorithm 3 Implementation of our approximate computing schemes within the parallel algorithm for community detection (Grappolo), shown for a single phase. The inputs are a graph ($G(V, E, \omega)$) and an array of size $|V|$ that represents an initial assignment of community for every vertex C_{init}. The output is the set of communities corresponding to the last iteration (with memberships projected back onto the original uncoarsened graph).

```
1:  procedure PARALLEL LOUVAIN($G(V, E, \omega), C_{init}$)
2:      $ColorSets \leftarrow Coloring(V)$, where $ColorSets$ represents
        a color-based partitioning of $V$.              ▷ An optional step
3:      $Q_{curr} \leftarrow 0$; $Q_{prev} \leftarrow -\infty$   ▷ Current & previous modularity
4:      $C_{curr} \leftarrow C_{init}$
5:      while true do
6:          for each $V_k \in ColorSets$ do
7:              $C_{prev} \leftarrow C_{curr}$
8:              for each $i \in Active(V_k)$ in parallel do
9:                  $N_i \leftarrow C_{prev}[i]$
10:                 for each $j \in \Gamma(i)$ do $N_i \leftarrow N_i \cup \{C_{prev}[j]\}$
11:                 $target \leftarrow \arg\max_{t \in N_i} \Delta Q_{i \to t}$
12:                 if $\Delta Q_{i \to target} > 0$ then
13:                     $C_{curr}[i] \leftarrow target$
14:
15:         $C_{set} \leftarrow$ set of communities corresponding to $C_{curr}$
16:         $Q_{curr} \leftarrow$ Compute modularity as defined by $C_{set}$
17:         if $|\frac{Q_{curr} - Q_{prev}}{Q_{prev}}| < \theta$ then      ▷ $\theta$ is a user specified
    threshold.
18:             break                                   ▷ Phase termination
19:         else
20:             $Q_{prev} \leftarrow Q_{curr}$
```

Algorithm 3, we show a function $Active(V_k)$ that returns only the subset of vertices that are still "active"—i.e., have *not* been terminated. To determine whether a vertex is to stay active during any given iteration, we use a binary flag at every vertex: 1, if it has changed its community affiliation in the last k iterations; and 0 otherwise. The flag is initialized to 1 at the start of execution. Once a vertex's flag becomes 0, it is "terminated".

Vertex Ordering via Graph Coloring: We employ graph coloring as a vertex ordering strategy for community detection. Distance-1 graph assigns a color to each vertex such that no two adjacent vertices are assigned the same color. We introduce the notion of incomplete coloring, where we color a subset of the vertices based on the number of colors that is provided as an input parameter (say k). To perform an incomplete coloring efficiently, we use the algorithm by Jones and Plussmann [16]. The algorithm starts by assigning a unique random number for each vertex. Unique colors are used for each iteration of the incomplete coloring algorithm. In a given iteration, each vertex checks to find if its random number is the maximum (alternatively, minimum) number among its neighbors (vertex identities can be used to break ties). A unique color is used to color all the locally maximum (alternatively, minimum) vertices (alternatively, minimum). This scheme guarantees that no two neighbors will receive the same color. The colored vertices are eliminated from further consideration after each iteration. The algorithm iterates until either all vertices are colored or the given number of colors specified has been exhausted. We call it incomplete since the number of colors used is small and potentially not sufficient to color all the vertices correctly. For such a situation, all the uncolored vertices are assigned a common color $k + 1$, and potentially with many conflicts (incorrect coloring). We experiment with $4, 8, 16$ and 32 colors in our experiments that are detailed in Section V.

Threshold Scaling: We implement threshold scaling in conjunction with coloring by using a higher value of threshold (θ in Algorithm 3) in the initial phases of the algorithm. In our experiments, we utilize a value of 10^{-2} as the higher threshold and 10^{-6} as the lower threshold. By decreasing the number of iterations in the beginning of the execution, the algorithm coarsens the graph quickly and thus converges faster. Empirically, we also observe that the final modularity is better when threshold scaling is combined with graph coloring as the heuristic [7].

V. Experimental Results and Observations

We evaluated the effectiveness of the approximate computing techniques that we employed for PageRank and community detection. We evaluated both algorithms using the same set of inputs on the same platform.

Experimental Setup: We tested our implementations on an 80-core Linux (kernel version 2.6.32) system consisting of eight ten-core Intel Xeon E7 8860 CPUs operating at 2.27GHz and two terabytes of total memory. We compiled the code with the Intel C++ Compiler version 15.0 and Intel Threading Building Blocks (TBB) version 4.4 using the '-Ofast' optimization flag. The graph datasets are from the Laboratory for Web Algorithmics (LAW) website [17]. We use two types of graphs for our experiments: social networks (twitter-2010 and hollywood-2011) and webgraphs (enwiki-2013, indochina-2004, uk-2002, eu-2015-host, arabic-2005, webbase-2001, it-2004, twitter-2010, sk-2005, uk-2007, gsh-2015 and clueweb12). Table I summarizes these inputs and their various characteristics. As shown, we group the inputs into three categories—smaller graphs that have less than 500 million edges, medium sized ones with less than 5 billion edges and two large graphs with greater than 30 billion edges. The same graph is interpreted as directed for PageRank and as undirected for community detection—a directed edge $u \to v$ is interpreted simply as an edge (u, v).

A. PageRank Evaluation

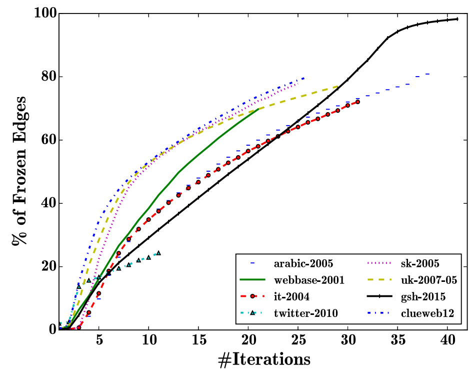
Now, we discuss the impact of our approximate computing strategy on the performance and accuracy of the PageRank results obtained. With the global threshold set to 10^{-6}, Table II shows the number of frozen edges for each dataset and the difference in global PageRank between the approximated (aka optimized) and baseline versions. The small difference (of up to $1.1e{-2}$) in the global PageRank shows that our approximate method minimally impacts the overall PageRank. The fraction of edges frozen keeps increasing until the final (converging) iteration as shown in Figure 2. The percentage improvements relative to the baseline version at various core counts are shown in Figure 3.

Table I
Input characteristics for the 14 datasets used in our experimental study. M and B denote millions and billions, respectively.

Input	No. Vertices	No. Edges	Out_Deg	
			Max.	Avg.
enwiki-2013	4.2M	101.4M	8104	24.1
indochina-2004	7.4M	194.1M	6985	26.1
hollywood-2011	2.2M	229.0M	13107	105.0
uk-2002	18.5M	298.1M	2450	16.1
eu-2015-host	11.3M	386.9M	398600	34.3
arabic-2005	22.7M	640.0M	9905	28.1
webbase-2001	118.1M	1.0B	3841	8.6
it-2004	41.3M	1.2B	9964	27.8
twitter-2010	41.7M	1.5B	3.0M	35.2
sk-2005	50.6M	1.9B	12870	38.5
uk-2007-05	105.9M	3.7B	15402	35.3
gsh-2015	988.5M	33.9B	32114	34.2
clueweb12	978.4M	42.6B	7447	43.5

Table II shows that, for all inputs, the number of iterations required for convergence remains nearly the same between the baseline and optimized versions. In general, we can say that the time per iteration improves for our approximate algorithm while keeping the number of iterations roughly the same for all datasets. The cumulative fraction of edges frozen along each iteration keeps increasing until the final iteration as shown in Figure 2. Hence, performance improves proportionally with the increasing number of frozen edges. We observe speedups of up to 2.3X.

Figure 2. Freezing rate across iterations.



From Figure 3, we observe that the performance improvements grow as graph sizes increase, with some exceptions like hollywood-2011, eu-2015-host, and twitter-2010. For these graphs, as shown in Table II, the total number of edges frozen is very small. Twitter-2010 and hollywood-2011 take very few iterations to converge and, hence, the number of frozen edges does not grow fast enough as their iterations progress. This behavior for twitter-2010 is shown in Figure 2.

In general, we observe the performance improvements from approximate computing are greater at smaller core

Table II
FROZEN COUNT, GLOBAL RANK DIFFS, ITERATION COUNT.

Dataset	#Edges	#Frozen	diff_global_rank	#Iter (base, approx)
enwiki-2013	101.4M	19.8M (19.49%)	3.3e-05	17, 17
indochina-2004	194.1M	107.3M (55.27%)	2.8e-04	31, 31
hollywood-2011	229.0M	376607 (0.16%)	0	6, 6
uk-2002	298.1M	198.0M (66.41%)	7.3e-04	26, 26
eu-2015-host	386.9M	108.1M (27.93%)	3.1e-04	36, 36
arabic-2005	640.0M	517.6M (80.88%)	8.0e-04	38, 38
webbase-2001	1.0B	711.4M (69.75%)	2.4e-03	21, 21
it-2004	1.2B	829.3M (72.07%)	1.3e-03	31, 31
twitter-2010	1.5B	357.5M (24.35%)	4.0e-05	11, 11
sk-2005	1.9B	1.5B (77.84%)	1.2e-03	28, 25
uk-2007-05	3.7B	2.9B (76.97%)	2.2e-03	29, 29
gsh-2015	33.9B	33.3B (98.24%)	1.1e-02	40, 41
clueweb12	42.6B	34.0B (79.94%)	9.5e-03	25, 26

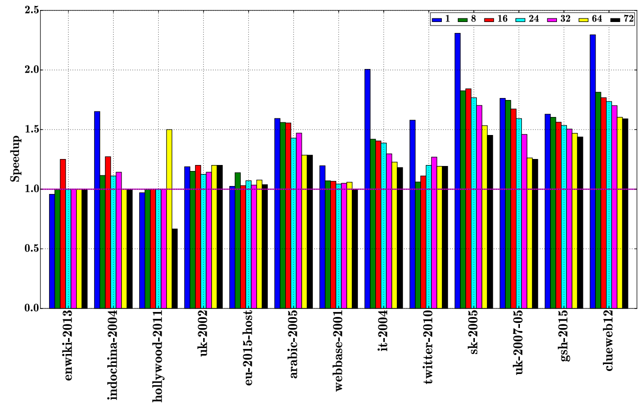


Figure 3. Speedup relative to baseline for 1.0e-6 global threshold.

counts than at larger core counts. At larger core counts, load balancing and other overheads limit the scalability achieved by the exact algorithm. The limited scalability and associated idleness potentially masks the benefits achieved from reduced memory operations.

We now analyze the accuracy of the results produced by approximate computing. The final PageRank vector produced in decreasing order of the final PageRank of each vertex. The L_1, L_2, and L_∞ norms presented in Table III are computed for all vertices in the graph. We observe that the L_2 and L_∞ norms are less than $7.4e{-}5$, demonstrating that the algorithm makes limited impact of the approximation on accuracy. The L_1 norm shows seemingly worse correctness bound of $1.1e{-}2$. Our approximate computing strategy accepts minor inaccuracies in lower ranked vertices, which are numerous in number. Together, these vertices have a significant impact on the L_1 norm. When we consider the 2000 highest ranked vertices, shown in Table IV, all errors, including the L_1 norms improve.

In general, the performance improvements resulting from approximate computing scale with lowering the convergence threshold. In scenarios where accuracy is important, our approach provides greater performance benefits without degrading the correctness of the search results.

1) Comparison with the STIC-D Algorithm: STIC-D [4] is a recent algorithm to optimize page rank computation through processing of vertices in terms of strongly connected components and selective freezing of vertices. It was shown to significantly improve performance as compared to prior alternatives. Working with the authors of STIC-D, we have implemented the STIC-D algorithm using our data structures and framework. We did not change the core logic implemented by the STIC-D authors, but instead optimized the primary data structures to make them consistent with our implementation. Specifically, our implementation of STIC-D uses contiguous cache aligned memory layout for the most important arrays, making our implementation 7-10X faster than the original version. We use this implementation of STIC-D to compare the accuracy and efficiency of our algorithm. Figure 5 shows the speedup of our optimized implementation compared relative to the STIC-D implementation. Figure 4 shows the L2 norms for both implementations. "OPT" in these figures refers to the optimized (i.e., approximate) version of our freezing-based algorithm. These results demonstrate that our optimized algorithm outperforms STIC-D for medium to large size datasets at scale in terms of both performance and correctness.

The difference between the algorithms is due to two reasons. STIC-D aggressively freezes computation by marking all contributions from a vertex as frozen. More importantly, the STIC-D algorithm computes the page rank one strongly connected component (SCC) at a time, enabling the freezing of contributions from other SCCs. This leads to better performance than our algorithm in certain cases at the expense of overall accuracy. On the other hand, computing in terms of SCCs reduces the overall parallelism available at any given time. This impacts the scalability achieved at large thread counts. Our approach operates on all vertices in parallel and selectively freezes edges (rather than vertices). This exposes greater parallelism and finer-grained control over the desired accuracy. In summary, the results demonstrate that STIC-D is an efficient algorithm for coarser accuracy requirements and at medium thread-counts. For larger thread counts and finer accuracy requirements, our algorithm performs better.

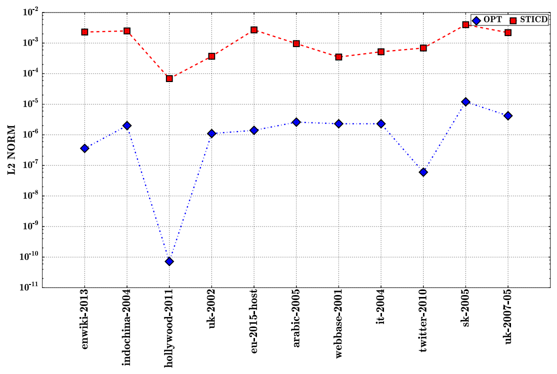


Figure 4. Comparison of L2 norms with STIC-D for 1.0e-6 global threshold

Table III
L_1, L_2 AND L_∞ NORMS (10^{-6}).

Dataset	L_1 Norm	L_2 Norm	L_∞ Norm
enwiki-2013	3.3e-05	3.6e-07	1.4e-07
indochina-2004	2.8e-04	2.1e-06	8.1e-07
hollywood-2011	1.6e-08	2.4e-10	4.5e-11
uk-2002	7.3e-04	1.4e-06	3.4e-07
eu-2015-host	3.1e-04	1.4e-06	6.0e-07
arabic-2005	8.0e-04	3.0e-06	1.2e-06
webbase-2001	2.4e-03	2.5e-06	9.4e-07
it-2004	1.3e-03	2.8e-06	4.2e-07
twitter-2010	4.6e-05	6.4e-08	3.9e-08
sk-2005	1.2e-03	1.2e-05	7.6e-06
uk-2007-05	2.2e-03	5.2e-06	9.7e-07
gsh-2015	1.1e-02	7.4e-05	4.3e-05
clueweb12	1.0e-02	5.2e-05	3.3e-05

Table IV
L_1, L_2 AND L_∞ NORMS FOR THE TOP 2000 RANKS WITH THRESHOLD 10^{-6}

Dataset	L_1 Norm	L_2 Norm	L_∞ Norm
enwiki-2013	5.8e-06	3.6e-07	1.4e-07
indochina-2004	4.3e-05	2.0e-06	8.1e-07
hollywood-2011	2.7e-10	7.2e-11	4.5e-11
uk-2002	3.2e-05	1.1e-06	3.4e-07
eu-2015-host	2.8e-05	1.4e-06	6.0e-07
arabic-2005	6.9e-05	2.6e-06	1.2e-06
webbase-2001	4.4e-05	2.3e-06	9.4e-07
it-2004	7.2e-05	2.3e-06	4.2e-07
twitter-2010	4.0e-07	6.0e-08	3.9e-08
sk-2005	2.3e-04	1.2e-05	7.6e-06
uk-2007-05	1.3e-04	4.2e-06	9.7e-07
gsh-2015	6.0e-04	7.4e-05	4.3e-05
clueweb12	7.5e-04	5.2e-05	3.3e-05

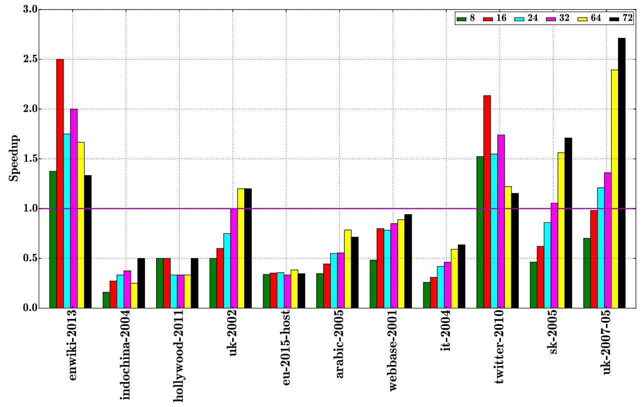


Figure 5. Speedup relative to STIC-D for 1.0e-6 global threshold

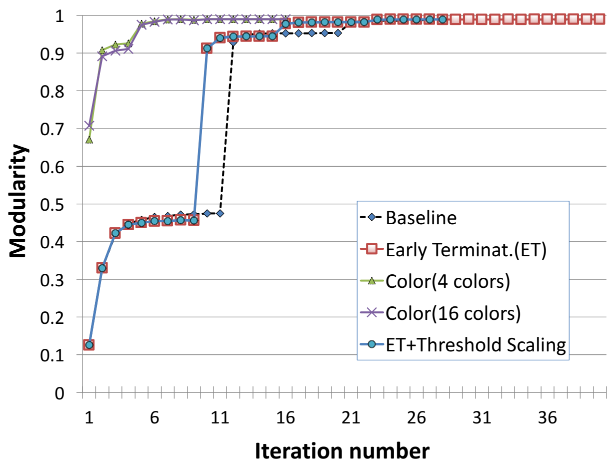


Figure 6. Impact of different techniques on the convergence of Grappolo for input `arabic-2005`.

B. Community Detection Evaluation

We interpret early termination as an approach for loop perforation for community detection, and incomplete coloring as the approach for vertex ordering. Further, threshold scaling and use of vectors are used as enabling techniques for the two basic approximate computing techniques. Different techniques impact the convergence of the algorithm, which we capture in Figure 6 for one input. We observe that faster convergence is enabled by the use of coloring combined with threshold scaling. While the use of vectors does not result in any improvement in the convergence of the algorithm, there is a significant improvement in the performance of each iteration, especially after the first few iterations when the number of communities decreases rapidly. We capture the details of convergence in terms of the modularity score at the end of the execution, the total number of iterations used and total number of phases employed in Table V.

We summarize the improvement in performance for Basic, Early Termination, Vector, and Early Termination combined with Threshold Scaling in Figure 7. We observe that while Early Termination results in the improvement of runtime for a given iteration, it has the effect of either increasing or decreasing the net number of iterations (e.g., increasing from 37 to 128 for IndoChina, while decreasing from 126 to 66 for uk-2002). We conclude that simple interpretation of loop perforation in a simple manner does *not* necessarily suffice for iterative algorithms such as community detection. A combination of Early Termination with Threshold Scaling leads to a better improvement in performance.

Table V
MODULARITY AND CONVERGENCE. "#ITS." DENOTES THE NUMBER OF ITERATIONS ACROSS ALL PHASES.

Input	**Baseline**		**Early Term.**		**Coloring (4)**	
	Modularity	#Its. (phases)	Modularity	#Its. (phases)	Modularity	#Its. (phases)
Arabic	0.989	35(5)	0.990	68(20)	0.990	16(6)
Wiki	0.671	58(5)	0.643	49(6)	0.633	19(5)
EU	0.828	46(5)	0.826	33(5)	0.828	16(5)
Hollywood	0.752	77(5)	0.735	37(7)	0.745	18(5)
IndoChina	0.929	37(6)	0.930	128(28)	0.930	17(6)
IT-2004	0.973	48(7)	0.973	44(7)	0.975	21(6)
sk-2205	0.971	24(4)	0.967	46(12)	0.967	12(4)
twitter-2010	0.478	31(6)	0.470	22(7)	0.444	32(5)
uk-2002	0.991	126(35)	0.990	66(15)	0.991	17(6)
uk-2007	0.996	43(6)	0.996	81(24)	0.994	21(6)
webbase	0.983	161(41)	0.983	114(34)	0.984	18(7)

The impact of coloring on the performance is summarized in Figure 8. We experimented with 4, 8, 16, and 32 colors and observe a similar behavior with each value. Notably,

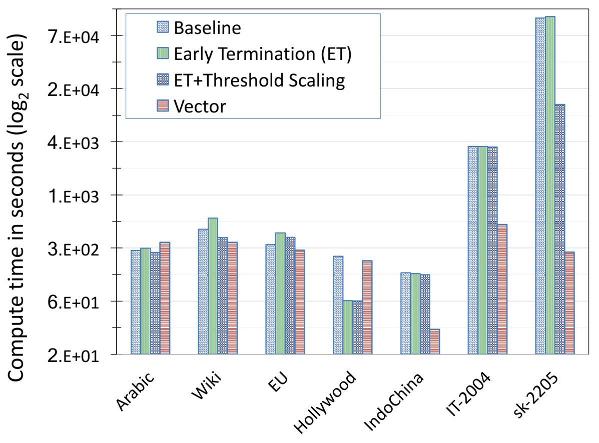


Figure 7. Impact of different approximate computing techniques on the performance of Grappolo. Different inputs are listed on the X-axis and compute time in seconds is provided on the Y-axis on a $\log_2$ scale.

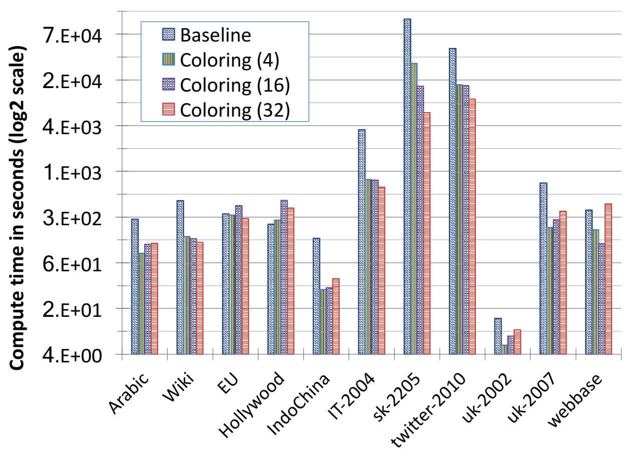


Figure 8. Impact of coloring with different colors on the performance of Grappolo. Different inputs are listed on the X-axis and compute time in seconds is provided on the Y-axis on a $\log_2$ scale.

the performance improvements due to coloring, compared to the baseline are significant while being widely varying—yielding up to $17\times$ for twitter-2010 and $450\times$ for sk-2205. As our next step, we are investigating the reason for this wide divergence in the speedups. As graph coloring takes only a small portion of the total time, one would be tempted to always use complete coloring. However, controlling the number of colors through an incomplete coloring could become important in distributed environments in order to reduce synchronization costs associated with each color class. Table VI shows the percentage of vertices colored by the different coloring schemes.

Table VI
% OF VERTICES COLORED FOR A GIVEN NUMBER OF COLORS

Input	4 colors (%)	8 colors (%)	16 colors (%)	32 colors (%)
Arabic	30.65	36.05	44.61	**56.17**
Wiki	25.75	29.57	32.72	**35.03**
EU	51.71	56.87	61.09	**64.23**
Hollywood	26.72	33.09	38.35	**41.62**
IndoChina	37.76	44.77	54.38	**66.05**
IT-2004	33.85	40.02	48.74	**59.53**
sk-2205	24.79	28.45	33.36	**40.13**
twitter-2010	34.89	37.58	39.42	**39.88**
uk-2002	38.42	46.68	59.08	**73.23**
uk-2007	34.75	38.63	45.03	**54.34**
webbase	53.63	65.42	78.18	**88.12**

VI. Related Work

PageRank is traditionally computed using the Power iteration method [3]. Since its inception, there has been a large body of work on optimizing PageRank using a variety of techniques. For instance, one well-known approach based on the power method is Adaptive PageRank [18], where the principal idea is that the pages that converge faster than other pages can be "locked" and not used in subsequent iterations leading to an overall 30% improvement. [19] studies the correlation between PageRank and indegree for webgraphs. Our approximation strategy is designed to be general enough to be applied to other types of graphs (social, citation, etc.) and not just webgraphs. [20] surveys approximation based approaches for computing PageRank.

Recent works using extrapolation-based methods to accelerate PageRank (e.g., [21], [22]) are based on the power method. They improve convergence rates but take more iterations compared to our exact version and do not provide any analysis on the correctness of the results obtained. Recently, D-Iteration [23] proposed a diffusion-based approach for solving PageRank as an alternative to the classical algorithms like the power method. While this approach shows very fast convergence rates using a 10^{-3} threshold, correctness is not evaluated. [24] demonstrates adaptive sampling methods for approximating PageRank; however our approach is faster in terms of convergence for similar datasets and is also evaluated on much larger datasets.

Community detection is also a well-studied problem with numerous related work. Due to the intractable nature of the problem [15], most of the widely used methods are efficient heuristics. [5] presents a detailed survey of the different approaches for community detection. [25] presents a survey of parallel implementations for community detection. Very recently, we had tested the early termination idea for reaping savings in performance and energy at the architecture layer, for Network-on-Chip manycore platforms [26]. However, the use of approximate computing techniques as proposed in this paper remains relatively an unexplored area. In fact, the techniques proposed in this paper have the potential to carry over to a broad range of graph algorithms with an iterative structure to their computation.

VII. Conclusion

The increasing need to perform graph analytics on large-scale datasets under strict constraints on time and energy can be effectively met with approximate computing techniques. Using PageRank and community detection as representative graph algorithms, we presented several techniques to improve performance with minimal impact on the quality of the solutions. We believe that the techniques we presented are applicable to a broad class of iterative graph algorithms. We conclude that approximate computing is an effective means to achieve desirable performance with a proportional impact on the loss of correctness and optimality of the solutions computed. However, the impact on quality of non-deterministic algorithms such as PageRank and Grappolo needs to be evaluated carefully. Metrics to quantify the quality of a solution also need to be chosen with care. Approximate computing holds promise to enable graph analytics on streaming data with restrictions on compute and memory capacities. We believe that the techniques presented in this paper will motivate their adoption to a diverse set of iterative algorithms by the community.

Acknowledgements

The research was supported in part by the DOD funded High Performance Data Analytics Program (HPDA) at the U.S. Department of Energy Pacific Northwest National Laboratory (PNNL), and by the DOE award DE-SC-0006516 to WSU. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. We thank the authors of STIC-D [4] for actively sharing the software and helping us port it to our framework. We also thank Dr. Hao Lu for his contributions to Grappolo.

References

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.

[2] 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 (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford University, Technical Report;, 1998.

[4] P. Garg and K. Kothapalli, "STIC-D: Algorithmic techniques for efficient parallel pagerank computation on real-world graphs," in *ICDCN*, 2016, pp. 15:1–15:10.

[5] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010.

[6] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008., 2008.

[7] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Comput.*, vol. 47, no. C, pp. 19–37, Aug. 2015.

[8] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 620–627, 2004.

[9] I. S. Duff, K. Kaya, and B. Uçcar, "Design, implementation, and analysis of maximum transversal algorithms," *ACM Trans. Math. Softw.*, vol. 38, no. 2, pp. 13:1–13:31, Jan. 2012.

[10] M. M. Ali Patwary, A. H. Gebremedhin, and A. Pothen, "New multithreaded ordering and coloring algorithms for multicore architectures," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 250–262.

[11] H. Lu, M. Halappanavar, D. Chavarria-Miranda, A. H. Gebremedhin, A. Panyala, and A. Kalyanaraman, "Algorithms for balanced graph colorings with applications in parallel computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1240–1256, 2017.

[12] U. Catalyurek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, vol. 38, no. 11, pp. 576–594, 2012.

[13] D. P. Bertsekas, "Auction algorithms for network flow problems: A tutorial introduction," *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, Oct 1992.

[14] D. F. Gleich, "PageRank beyond the web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, August 2015.

[15] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE transactions on knowledge and data engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[16] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 654–669, May 1993.

[17] "Laboratory for web algorithmics (law)," http://law.di.unimi.it/.

[18] S. Kamvar, T. Haveliwala, and G. Golub, "Adaptive methods for the computation of pagerank," *Linear Algebra and its Applications*, vol. 386, pp. 51 – 65, 2004.

[19] S. Fortunato, M. Boguñá, A. Flammini, and F. Menczer, "Algorithms and models for the web-graph," 2008, ch. Approximating PageRank from In-Degree, pp. 59–71.

[20] F. Chung, "A brief survey of pagerank algorithms," *IEEE Transactions on Network Science and Engineering*, vol. 1, no. 1, pp. 38–42, Jan 2014.

[21] H.-F. Zhang, T.-Z. Huang, C. Wen, and Z.-L. Shen, "{FOM} accelerated by an extrapolation method for solving pagerank problems," *Journal of Computational and Applied Mathematics*, vol. 296, pp. 397 – 409, 2016.

[22] Y.-J. Xie and C.-F. Ma, "A relaxed two-step splitting iteration method for computing pagerank," *Computational and Applied Mathematics*, pp. 1–13, 2016.

[23] D. Hong, T. D. Huynh, and F. Mathieu, "D-iteration: diffusion approach for solving pagerank," *CoRR*, vol. abs/1501.06350, 2015.

[24] W. Liu, G. Li, and J. Cheng, "Fast pagerank approximation by adaptive sampling," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 127–146, Jan. 2015.

[25] A. Kalyanaraman, M. Halappanavar, D. Chavarría-Miranda, H. Lu, K. Duraisamy, P. P. Pande *et al.*, "Fast uncovering of graph communities on a chip: Toward scalable community detection on multicore and manycore platforms," *Foundations and Trends® in Electronic Design Automation*, vol. 10, no. 3, pp. 145–247, 2016.

[26] K. Duraisamy, H. Lu, P. P. Pande, and A. Kalyanaraman, "Accelerating graph community detection with approximate updates via an energy-efficient NoC," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 89.