Scalable Static and Dynamic Community Detection Using Grappolo

Mahantesh Halappanavar[1], Hao Lu[2], Ananth Kalyanaraman[3], and Antonino Tumeo[1]

E-mail: hala@pnnl.gov, luh1@ornl.gov, ananth@eecs.wsu.edu, antonino.tumeo@pnnl.gov
[1]Pacific Northwest National Laboratory [2]Oak Ridge National Laboratory [3]Washington State University.

***Abstract*—Graph clustering, popularly known as community detection, is a fundamental kernel for several applications of relevance to the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit (HIVE) Program. Clusters or communities represent natural divisions within a network that are densely connected within a cluster and sparsely connected to the rest of the network. The need to compute clustering on large scale data necessitates the development of efficient algorithms that can exploit modern architectures that are fundamentally parallel in nature. However, due to their irregular and inherently sequential nature, many of the current algorithms for community detection are challenging to parallelize. In response to the HIVE Graph Challenge, we present several parallelization heuristics for fast community detection using the *Louvain* method as the serial template. We implement all the heuristics in a software library called Grappolo. Using the inputs from the HIVE Challenge, we demonstrate superior performance and high quality solutions based on four parallelization heuristics. We use Grappolo on static graphs as the first step towards community detection on streaming graphs.**

I. Introduction

In response to the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit Program (HIVE) Graph Challenge, we submit our work in the broad category of graph clustering for static and dynamic graphs [1]. In this work, we focus on static graphs as the first step towards dynamic graphs. We present several heuristics to enable parallelization of an inherently serial algorithm, and demonstrate that these heuristics improve the overall quality of computed solutions.

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 [2]. We employ several heuristics to improve the performance of an agglomerative technique based on modularity optimization [3], the Louvain algorithm. We present our work on parallelizing the widely used Louvain algorithm through a set of heuristics that not only enable parallelization, but also improve the quality of solutions. Our heuristics use approximate computing to explore and derive trade-offs between performance and quality. We present the details in Section II.

We define a dynamic graph as a graph that changes over time. Changes can include vertex (node) and edge (link) addition and deletion. A snapshot (or time slice) of this graph, G_i, consists of the vertices and edges active at a given timestep i. Modifications from time i to $i + 1$ are represented by ΔG_i.

A community, $C(G)$, in graph G represents a subset of vertices. Similar to the evolution of a graph, the communities also evolve. Temporal communities can have several operations: growth (via addition of new nodes), contraction (via elimination of nodes), merging (of two or more communities), splitting (into two or more communities), birth (of a new community), death, and resurgence (reappearance after a period of time).

We develop two approaches for dynamic community detection—one that computes the communities at each timestep and another that allows a systematic propagation of communities from the previous snapshot to the current snapshot. The two steps involved in the latter approach are: static community detection in the first snapshot (G_0), and propagation of communities between two snapshots (G_i and G_{i+1}) by simultaneously optimizing the quality of $C(G_{i+1})$ and its similarity to $C(G_i)$. We detail our approach in Section III.

We present experimental results from the execution of our algorithm on the HIVE Challenge dataset in Section IV. We present results on the quality as well as performance in this section.

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

- Performance evaluation and empirical validation of the correctness of the solutions using the HIVE datasets with ground truth.
- Design and evaluation of multiple heuristics for parallelization of community detection, with a potential to extend to other iterative graph codes; and
- Presentation of techniques for the application of scalable community detection on static graphs in the context of dynamic graphs.

A. Related Work

Community detection is an active area of research. We refer the reader to [2], [4] for an extensive review of the topic. The seminal work by Newman and Girvan in introducing the modularity metric [5], motivated the development of divisive [5], [6] as well as agglomerative [7] clustering methods. While a divisive method uses betweenness centrality to identify and remove bridges between communities, agglomerative clustering approach greedily

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

merges two communities that provide the maximum gain in modularity. Analytically and practically, agglomerative methods are faster than divisive, but suffer from several limitations. The Louvain method [3] is a variant of the agglomerative strategy, in that it is a multi-level heuristic, and within each level it enables vertices to make decisions independently from their current community assignments at each step.

There also exists a large body of work on parallelizing community detection algorithms. In [8], we presented an extensive survey of the state of parallel methods for community detection. Notable among these works are as follows. As part of the DIMACS10 clustering challenge, Riedy *et al.* presented a highly parallel agglomerative implementation [9], [10] for the Clauset-Newman-Moore (CNM) algorithm [7]. Auer and Bisseling [11] present another way to achieve agglomerative clustering on GPUs using graph coarsening. LaSalle and Karypis [12] present a multilevel graph clustering method for shared memory machines. Recently, Naim *et al.,* presented their efforts on parallelizing the Louvain method on GPUs [13].

II. Parallel Heuristics 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 $P = \{C_1, C_2, \ldots C_k\}$ 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 [5]:

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

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 [14].

The Louvain algorithm proposed by Blondel *et al.* [3] 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.* [15]. 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 [15] 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 1. Within each iteration, vertices are considered in parallel (Line 9) 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 (Line 7), 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 11-13).

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). Our implementation is named *Grappolo*. A preliminary version of the software is available for download under the BSD 3-Clause license at: http://hpc.pnl.gov/people/hala/grappolo.html.

A. Heuristics for parallelization

We employ four different techniques for parallelization: (i) Vertex following and minimum label heuristic, (ii) data caching, (iii) graph coloring, and (iv) threshold scaling. We briefly explain each of these heuristics in the following discussion.

Vertex following and Minimum Label heuristics: Many real world graphs contain a large number of single-degree vertices. It is easy to observe that there is no need to explicitly make decisions on these vertices during an iteration of the Louvain algorithm. We therefore preprocess the input and merge all single-degree vertices with their corresponding neighbors. We make a distinction between singleton nodes and edges, and single-degree vertices. The neighbor of a single-degree vertex is also not a single-degree vertex. We remove the single-degree vertices by adding a self-edge to their respective neighbors and set the weight of the self-edge to the weight of the edge that is removed. The

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Algorithm 1 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).

procedure PARALLEL LOUVAIN($G(V, E, \omega), C_{init}$)
  $ColorSets \leftarrow Coloring(V)$, where $ColorSets$ represents a color-based partitioning of V. ▷ An optional step
  $Q_{curr} \leftarrow 0$; $Q_{prev} \leftarrow -\infty$ ▷ Current & previous modularity
  $C_{curr} \leftarrow C_{init}$
  while `true` **do**
    for each $V_k \in ColorSets$ **do**
      $C_{prev} \leftarrow C_{curr}$
      for each $i \in Active(V_k)$ in `parallel` **do**
        $N_i \leftarrow C_{prev}[i]$
        for each $j \in \Gamma(i)$ **do** $N_i \leftarrow N_i \cup \{C_{prev}[j]\}$
        $target \leftarrow \arg\max_{t \in N_i} \Delta Q_{i \to t}$
        if $\Delta Q_{i \to target} > 0$ **then**
          $C_{curr}[i] \leftarrow target$

    $C_{set} \leftarrow$ set of communities corresponding to C_{curr}
    $Q_{curr} \leftarrow$ Compute modularity as defined by C_{set}
    if $|\frac{Q_{curr} - Q_{prev}}{Q_{prev}}| < \theta$ **then** ▷ θ is a user specified threshold.
      `break` ▷ Phase termination
    else
      $Q_{prev} \leftarrow Q_{curr}$

single-degree vertices are assigned the same community that their neighbors get assigned at the end of execution. This preprocessing not only helps reduce the number of vertices that need to be considered during each iteration, but also enables the vertices that have many single-degree neighbors (hubs) to be the seeds of community migration decisions. This becomes especially important in a parallel context.

For a given iteration of Algorithm 1, a vertex v can have multiple neighboring communities yielding the same (maximum) modularity gain. We use the *minimum label* heuristic to make a decision by selecting the minimum label among the available neighboring communities as the destination for i's new community. This simple heuristic tends to minimize or prevent community swaps and local maxima [15]. We employ vertex following and minimum labeling in all of our experiments presented in the paper.

Data caching: Within each iteration of Algorithm 1, a vertex considers all the available communities to join and chooses the one with a maximum gain. In order to store this information, we can employ ordered or unordered maps. However, the use of map data structure can lead to excessive memory allocation and deallocation costs along with irregular memory access patterns. We therefore, replace the map data structure with a vector and reuse the memory for each iteration, but with an additional cost to compare the existing entries. Empirically, we observe that the benefits

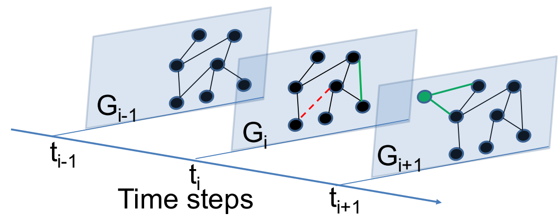

Figure 1. Schematic illustration of a dynamic graph.

of replacing the map data structure can lead to significant gains in performance (up to $10\times$) when the number of communities decreases rapidly. However, in some cases, it can lead to loss of performance when the number of communities remains large for many iterations (inputs that converge slowly). We enable the data caching heuristic in all of our experiments presented in the paper.

Vertex Ordering via Graph Coloring: A distance-k coloring of a graph is an assignment of colors (unique integers) to vertices such that no two vertices at a distance of k hops are assigned the same color. As a consequence, distance-1 coloring will generate vertex partitions such that no two vertices in the same set are neighbors of each other. As presented in Algorithm1, we process each vertex set concurrently and synchronize after each color class. As demonstrated in [15], parallel execution in this manner tends to mimic the behavior of a serial algorithm in terms of the gain in modularity per iteration.

Threshold Scaling: Threshold scaling is the idea of using a successively smaller threshold value (θ in Algorithm 1) as the algorithm progresses. In our experiments, we utilize a value of 10^{-2} as the higher threshold value, and 10^{-6} as the lower threshold value. We employ threshold scaling in conjunction with the coloring heuristic by using a higher value of threshold in the initial phases of the algorithm, and a lower threshold value towards the end of the execution. Empirically, we also observe that the algorithm converges faster and evolves towards a better modularity score when threshold scaling is combined with graph coloring [15]. We present results from all the four heuristics in Section IV.

III. Community Detection on Dynamic Graphs

Our algorithm for community detection on dynamic graphs uses the following model of dynamic graphs (see Figure 1 for an illustration). Let $G_i(V_i, E_i)$ denote a graph observed at timestep i, where $i \in [1, t]$. Graph edits from one timestep to the next occur in the following forms:

- **edge addition:** a new edge gets added between two vertices (old or new);
- **edge deletion:** an existing edge gets removed between two existing vertices.

We implement two versions of our algorithm for identifying communities of a dynamic graph.

- **Unseeded clustering:** In this scheme, we treat the graph at each timestep as an independent instance and

- run Grappolo on it. The advantage of this scheme is that all changes and their full impacts on clustering are implicitly taken into account while performing the clustering at each step. A potential disadvantage, however, is in the performance—i.e., the cost of recomputing the clustering on the entire graph regardless of how localized and sparse the graph edits may be.
- **Seeded clustering:** In this scheme, we try to propagate the community information from the previous timestep, in the current timestep. More specifically, we initialize the set of vertices in G_i to their community states at the end of timestep $(i-1)$. Subsequently, the Grappolo algorithm is run on G_i. The advantage of this scheme is potentially faster convergence, in that if the graph edits are highly localized and incremental in nature, the Grappolo algorithm is likely to converge in very few iterations compared to the seeded version. This scheme is also better prepared to propagate community information from across timesteps, thereby facilitating community tracking. A potential disadvantage of this scheme, however, is that of biasing—i.e., the community configuration reached at the end of timestep $(i-1)$ may be suboptimal as a starting point for the kinds of edits that have accrued in timestep i.

The seeded and unseeded clustering schemes offer a trade-off in quality and performance.

IV. Experimental Results

In this section, we present results from the empirical evaluation of Grappolo using the HIVE Challenge datasets. We evaluate Grappolo using two configurations: (i) *Basic:* where we enable Vertex Following, Minimum Label and Data caching heuristics, and (i) *Advanced:* where we enable all the previous heuristics along with Coloring and Threshold Scaling heuristics.

All the experiments were executed on a shared-memory system with two 10-core Xeon CPU E5-2680 v2 processors operating at 2.80GHz. We disabled HyperThreading, so each processor supports up to 10 physical threads. Each processor has 25 MB of L3 cache, and the system has 768 GB of DDR3 memory. We used Redhat linux operating system with Kernel 2.6.32 and compiled our code with GCC 4.9.2 using the '-Ofast' optimization flag. To compute performance metrics, we used the snap datasets of the DARPA HIVE Graph Challenge.

Qualitative Assessment: In order to assess the quality of computed solutions, we use the following metrics. Consider two community assignments C_T and C_O, where C_T represents the community assignment provided as ground truth, and C_O is the community assignment as computed by Grappolo. We consider all possible pairs of vertices in C_T and C_O and categorize each pair (u, v) into one of the three following bins:

- **True Positive (TP) or Same-Same:** if u and v belong to the same community in both assignments;
- **False Negative (FN) or Same-Diff:** if u and v belong to the same community only in assignment C_T; or
- **False Positive (FP) or Diff-Same:** if u and v belong to the same community only in assignment C_O;

Based on the above categorization, we define the following metrics:

- **Precision:** is given by the ratio: $P = \frac{TP}{TP+FP}$;
- **Recall:** is given by the ratio: $R = \frac{TP}{TP+FN}$;
- **F-Score:** is given by the ratio: $F = 2.\frac{P.R}{P+R}$;

We summarize the results on the large set of inputs with ground truth in Table I. We note that the metrics for precision and recall is 100% for all the small inputs. We observe that Grappolo computes high quality solutions for the four instances with ground truth. We plan to perform detailed comparisons with a large number of inputs using the reference implementation provided through the HIVE Challenge and other instances with ground truth information.

Performance Analysis: We summarize detailed information for Basic and Advanced variants of Grappolo in Table II. For each input, we present runtimes using 2, 10 and 20 threads for both the variants. We also present the information on the total number of iterations and the modularity values at the end of the execution.

We observe that a large number of inputs from the challenge datasets are small in size, and consequently, we do not observe meaningful speedups with larger number of threads. We refer you to [15] for a detailed analysis of Grappolo with larger inputs. We highlight the differences between the Basic and Advanced variants through performance and modularity values in Figures 2 and 3. We note that this difference in performance is driven by coloring and threshold scaling heuristics. We plan to extend this analysis to a set of larger inputs and include experimental results for dynamic community detection.

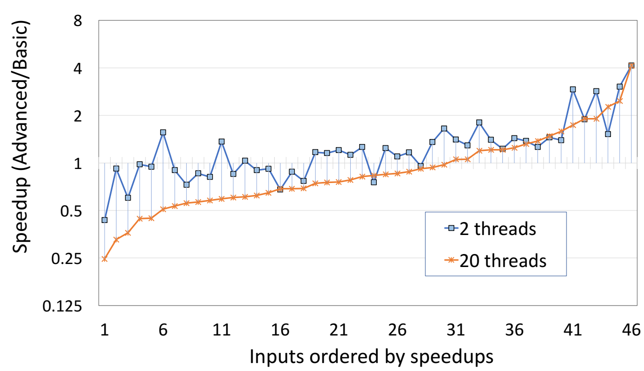

Figure 2. Speedup of Advanced relative to Basic for all the inputs from Table II. For small inputs, we observe runtime variations due to various reasons. We executed multiple runs and selected one particular set of runs for reporting in this paper. The inputs are ordered based on the speedup values.

V. Conclusion

Performing community detection on streaming data necessitates the development of efficient algorithms that can

Table I
QUALITY COMPARISONS WITH GROUND TRUTH COMMUNITY INFORMATION.

			Basic						Advanced					
Input	$\lvert V\rvert$	$\lvert E\rvert$	**#Itrs**	**Modularity**	**Time(s)**	**Precision**	**Recall**	**F-Score**	**#Itrs**	**Modularity**	**Time(s)**	**Precision**	**Recall**	**F-Score**
simulated_blockmodel_graph_20k	2E+04	4.09E+05	14	0.793	0.20	100.00%	100.00%	1.00	9	0.793	0.20	100.00%	100.00%	1.00
simulated_blockmodel_graph_50k	5E+04	1.02E+06	24	0.806	0.37	100.00%	100.00%	1.00	8	0.806	0.38	100.00%	100.00%	1.00
simulated_blockmodel_graph_2M	2E+06	4.07E+07	16	0.700	20.04	80.94%	81.03%	0.81	11	0.830	7.11	70.29%	100.00%	0.83
simulated_blockmodel_graph_5M	5E+06	2.30E+08	12	0.645	111.20	84.01%	84.13%	0.84	11	0.799	40.83	100.00%	100.00%	1.00

Table II
INFORMATION FROM THE EXECUTION OF GRAPPOLO USING TWO VERSIONS – BASIC AND ADVANCED.

Input	$\lvert V\rvert$	$\lvert E\rvert$	Basic					Advanced				
			2T	**10T**	**20T**	**#Itrs**	**Modularity**	**2T**	**10T**	**20T**	**#Itrs**	**Modularity**
:amazon0302	262111	899792	1.594030	0.575909	0.407473	109	0.899163	0.545578	0.233194	0.234519	25	0.899357
amazon0312	400727	2349869	3.355239	1.253326	0.839243	96	0.873400	1.177460	0.509870	0.439975	17	0.873581
amazon0505	410236	2439437	2.337863	1.058725	0.723567	66	0.871726	1.236590	0.526230	0.379453	20	0.873681
amazon0601	403394	2443408	3.434897	1.234979	0.850175	91	0.874345	1.129834	0.530699	0.344100	21	0.876185
as20000102	6474	12572	0.007792	0.006767	0.004445	19	0.620866	0.007955	0.007762	0.010004	16	0.586251
as-caida20071105	26475	53381	0.042227	0.029995	0.023517	28	0.661544	0.036205	0.024553	0.026668	12	0.665536
ca-AstroPh	18772	198050	0.143864	0.040344	0.023732	47	0.624232	0.092194	0.045287	0.046514	14	0.618623
ca-CondMat	23133	93439	0.071300	0.024234	0.019236	41	0.723343	0.061767	0.027177	0.025508	20	0.729553
ca-GrQc	5242	14484	0.005782	0.005842	0.003798	26	0.859141	0.013306	0.013273	0.015368	15	0.858567
ca-HepPh	12008	118489	0.046765	0.034122	0.031866	32	0.660030	0.077768	0.070059	0.088119	22	0.654684
ca-HepTh	9877	25973	0.022064	0.009812	0.010530	40	0.766525	0.021360	0.013813	0.017241	16	0.762862
cit-HepPh	34546	420877	0.294538	0.092860	0.085235	54	0.724256	0.163376	0.070566	0.071163	17	0.716797
cit-HepTh	27770	352285	0.241075	0.072747	0.062652	49	0.655739	0.145852	0.069717	0.064201	23	0.651877
cit-Patents	3774768	16518947	71.402024	22.681512	16.311464	118	0.805231	17.281550	6.631884	3.950743	23	0.804871
email-Enron	36692	183831	0.121916	0.053130	0.042008	39	0.618508	0.089303	0.052349	0.070903	18	0.608604
email-EuAll	265214	364481	0.173028	0.123556	0.101028	29	0.788428	0.113796	0.064195	0.044519	14	0.792712
facebook_combined	4039	88234	0.022509	0.011155	0.013245	28	0.834956	0.024431	0.023541	0.040468	12	0.834579
flickrEdges	105938	2316948	1.409969	0.992169	0.893408	51	0.674564	1.870223	0.949590	1.071839	20	0.671482
loc-brightkite_edges	58228	214078	0.177203	0.078181	0.069497	42	0.686430	0.126521	0.066562	0.057419	19	0.682115
loc-gowalla_edges	196591	950327	0.953243	0.788969	0.698657	39	0.697469	0.663912	0.525380	0.559011	21	0.712544
oregon1_010331	10670	22002	0.013333	0.010447	0.009839	26	0.613603	0.016347	0.014489	0.017004	19	0.629268
oregon1_010407	10729	21999	0.012964	0.011081	0.009344	24	0.619142	0.016856	0.013539	0.013532	14	0.624712
oregon1_010414	10790	22469	0.013147	0.009917	0.009392	21	0.615229	0.018040	0.015774	0.016853	20	0.619420
oregon1_010421	10859	22747	0.017085	0.014096	0.014450	35	0.616592	0.013207	0.013628	0.013694	18	0.621334
oregon1_010428	10886	22493	0.012233	0.009421	0.010540	21	0.600536	0.013919	0.011093	0.015346	11	0.618082
oregon1_010505	10943	22607	0.012686	0.010573	0.012034	25	0.603981	0.013263	0.011588	0.013072	17	0.623284
oregon1_010512	11011	22677	0.015015	0.010853	0.010742	25	0.608212	0.013349	0.011321	0.013739	12	0.622303
oregon1_010519	11051	22724	0.012360	0.009765	0.009636	22	0.603667	0.014382	0.012270	0.016990	12	0.616358
oregon1_010526	11174	23409	0.015124	0.011224	0.012906	25	0.615469	0.016460	0.015542	0.019895	13	0.620432
oregon2_010331	10900	31180	0.022094	0.014633	0.013782	31	0.646424	0.018893	0.013589	0.018552	16	0.643897
oregon2_010407	10981	30855	0.017673	0.011931	0.007927	28	0.638002	0.018679	0.014505	0.017735	16	0.638714
oregon2_010414	11019	31761	0.017283	0.010713	0.010796	25	0.625981	0.019196	0.018815	0.017356	16	0.558788
oregon2_010421	11080	31538	0.017658	0.012677	0.010512	30	0.630238	0.019607	0.016914	0.019666	19	0.633541
oregon2_010428	11113	31434	0.022977	0.015975	0.014306	31	0.632969	0.018480	0.014703	0.016873	9	0.599262
oregon2_010505	11157	30943	0.029587	0.018526	0.017147	36	0.637232	0.021033	0.019310	0.016267	17	0.633437
oregon2_010512	11260	31303	0.013736	0.010196	0.010874	23	0.637165	0.020254	0.015834	0.015848	17	0.642299
oregon2_010519	11375	32287	0.023924	0.014476	0.011805	35	0.628060	0.019832	0.016446	0.015550	17	0.630687
oregon2_010526	11461	32730	0.018611	0.014038	0.011221	28	0.627609	0.021887	0.019636	0.018576	19	0.633561
p2p-Gnutella04	10876	39994	0.043061	0.023122	0.020681	31	0.322036	0.031760	0.025478	0.022077	21	0.376288
p2p-Gnutella05	8846	31839	0.034872	0.022107	0.027617	24	0.340746	0.024005	0.020490	0.018676	18	0.394400
p2p-Gnutella06	8717	31525	0.030688	0.020862	0.017812	24	0.349240	0.027925	0.015605	0.020742	25	0.375117
p2p-Gnutella08	6301	20777	0.020831	0.015390	0.019042	23	0.395347	0.016951	0.010009	0.015693	21	0.448751
p2p-Gnutella09	8114	26013	0.024316	0.010490	0.013514	25	0.408978	0.019267	0.017100	0.016452	21	0.456839
p2p-Gnutella24	26518	65369	0.079952	0.043956	0.033449	24	0.440771	0.057971	0.025718	0.025374	22	0.456986
p2p-Gnutella25	22687	54705	0.063548	0.032649	0.030620	26	0.469230	0.050276	0.027388	0.022270	19	0.488688
p2p-Gnutella30	36682	88328	0.103806	0.038407	0.060773	29	0.480391	0.074518	0.038286	0.038340	22	0.507539
p2p-Gnutella31	62586	147892	0.181322	0.094135	0.084359	28	0.467652	0.124112	0.066190	0.065554	20	0.491227
roadNet-CA	1965206	2766607	1.152737	0.427426	0.857121	54	0.991811	1.849306	0.973869	0.891868	27	0.992137
roadNet-PA	1088092	1541898	0.708487	0.576023	0.477863	53	0.988694	0.999466	0.550691	0.360669	26	0.989183
roadNet-TX	1379917	1921660	1.005758	0.538990	0.601432	54	0.990795	1.156458	0.698840	0.643208	26	0.991194
soc-Epinions1	75879	405740	0.435763	0.199228	0.159041	49	0.447397	0.213260	0.129149	0.137088	22	0.449320
soc-Slashdot0811	77360	469180	0.255844	0.162006	0.120686	23	0.280334	0.257484	0.171831	0.197298	23	0.340628
soc-Slashdot0902	82168	504230	0.297483	0.171952	0.153559	23	0.259748	0.291204	0.161546	0.169532	21	0.342392
friendster	119432957	1799999986	-	-	2519.693423	38	0.471124	-	-	812.742337	20	0.556225

also exploit modern computer architectures. Towards this end, we presented several heuristics for parallelization of the Louvain algorithm and demonstrated their effectiveness using the datasets from the DARPA HIVE Graph Challenge. Our goal was to address the dual objectives of maximizing concurrency while improving the quality with respect to the serial implementation. We also presented our current approach to extend the work on static graphs to enable efficient community detection on dynamic graphs.

We plan to extend Grappolo in two directions in the near future: (i) distributed-memory implementations, and (ii) community detection on streaming graphs. We have preliminary work in both of these areas with promising results. Our current work on distributed-memory implementations includes a high performance implementation using MPI and OpenMP with incomplete coloring and threshold scaling heuristics. Our preliminary work on dynamic graphs includes methods to track communities as well as to efficiently seed community detection method.

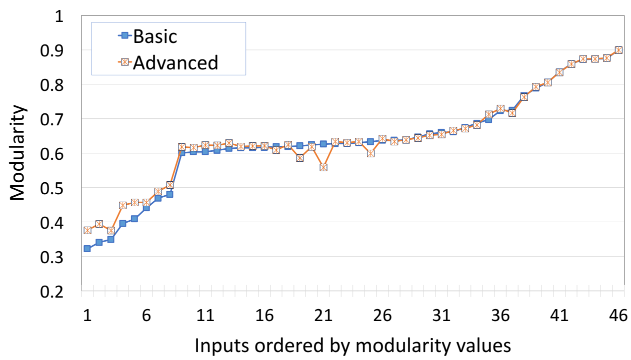

Figure 3. Modularity values for the two variants of the algorithm – Basic and Advanced – for all the inputs from Table II. The inputs are ordered based on the modularity values.

Acknowledgements

The research is in part supported by the U.S. Department of Energy's (DOE) Exascale Computing Project (ExaGraph), the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit Program and the High Performance Data Analytics Program (HPDA) at DOE Pacific Northwest National Laboratory (PNNL), and by DOE award DE-SC-0006516 to WSU. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith, "Streaming Graph Challenge: Stochastic Block Partition," in *IEEE HPEC*, 2017.

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

[3] 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.

[4] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, pp. 167–256, 2003.

[5] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, 2004.

[6] M. E. J. Newman, "Analysis of weighted networks," *Phys. Rev. E*, vol. 70, no. 5, p. 056131, Nov. 2004.

[7] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, no. 6, p. 066111, Dec. 2004.

[8] A. Kalyanaraman, M. Halappanavar, D. Chavarría-Miranda, H. Lu, K. Duraisamy, and P. Pratim Pande, "Fast uncovering of graph communities on a chip: Toward scalable community detection on multicore and manycore platforms," *Found. Trends Electron. Des. Autom.*, vol. 10, no. 3, pp. 145–247, Aug. 2016.

[9] J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 1619–1628.

[10] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Parallel Processing and Applied Mathematics*. Springer, 2012, pp. 286–296.

[11] B. F. Auer and R. H. Bisseling, "Graph coarsening and clustering on the GPU," *Graph Partitioning and Graph Clustering*, vol. 588, p. 223, 2012.

[12] D. LaSalle and G. Karypis, "Multi-threaded modularity based graph clustering using the multilevel paradigm," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 66–80, 2015.

[13] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo, "Community detection on the gpu," in *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017.

[14] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 20, no. 2, pp. 172–188, 2008.

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

978-1-5386-3472-1/17/$31.00 ©2017 IEEE