Scalable Distributed Memory Community Detection Using Vite

Sayan Ghosh*, Mahantesh Halappanavar†, Antonino Tumeo†, Ananth Kalyanaraman* Assefaw H. Gebremedhin*

* Washington State University, Pullman, WA, USA {sayan.ghosh, ananth, assefaw.gebremedhin}@wsu.edu
† Pacific Northwest National Laboratory, Richland, WA, USA {hala, antonino.tumeo}@pnnl.gov

***Abstract*—Graph clustering, popularly known as community detection, is a fundamental graph operation used in many applications related to network analysis and cybersecurity. The goal of community detection is to partition a network into "communities" such that each community consists of a tightly-knit group of nodes with relatively sparser connections to the rest of the nodes in the network. To compute clustering on large-scale networks, efficient parallel algorithms capable of fully exploiting features of modern architectures are needed. 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 2018 Streaming Graph Challenge, we present Vite—a distributed memory parallel implementation of the *Louvain* method, a widely used serial method for community detection. In addition to a baseline parallel implementation of the Louvain method, Vite also includes a number of heuristics that significantly improve performance while preserving solution quality. Using the datasets from the 2018 Graph Challenge (static and streaming), we demonstrate superior performance and high quality solutions.**

I. Introduction

In response to the 2018 Graph Challenge [1], we present our work in the broad category of graph clustering for static and streaming graphs.

Given an undirected graph $G = (V, E, \omega)$, where V is the vertex set, E is the edge set and ω represents edge weights, community detection aims to compute a partitioning of V into a set of tightly-knit communities (or clusters). Community detection is among the most frequently used graph structure discovery tools in a network scientist's toolkit [2]. Thanks to the rapid advancement of high throughput data generation and sensing techniques across scientific and industrial domains, real-world networks constructed from raw data are becoming extremely large and complex to analyze.

In this paper, we present empirical evaluation of Vite—a distributed memory implementation of a parallel community detection method [3]—on the 2018 Graph Challenge inputs. Vite is a parallel implementation of the Louvain method, which is a widely used sequential heuristic for community detection based on modularity optimization [4]. In addition to parallelizing the algorithm for distributed memory parallel computers, we have implemented multiple heuristics to further exploit the graph structure, expose parallelism, and take advantage of certain properties of the underlying algorithm. Broadly, these heuristics involve two techniques: (i) *graph coloring* to generate a partial ordering of vertices; and (ii) *approximate computing* for exploiting convergence properties of the algorithm for the purpose of performance-quality tradeoffs. The algorithmic details are presented in §II.

In §III, we present a detailed experimental evaluation of Vite (and configurations of its heuristics) on the 2018 Graph Challenge datasets. We analyze both performance and quality of the outputs generated by Vite. Primary experimental evaluations were performed on NERSC Edison supercomputer, with up to 1.5K processes. We also use NERSC Cori supercomputer (which uses the same network interconnect as NERSC Edison, with newer processors) to demonstrate scalability up to 4K processes for some cases. The experimental results discussed in §III show that our method is able to deliver excellent parallel performance and reduce time-to-solution while preserving quality. Vite is available for download under the BSD 3-clause license from: http://hpc.pnl.gov/people/hala/grappolo.html.

II. Parallel algorithm

In this section, we present a high level overview of Vite, our distributed memory implementation for parallel community detection, and the different heuristics it implements. For a detailed description, we refer the reader to [3].

A. Overview of Vite

Vite is a parallel implementation of the well known Louvain algorithm [4], which uses modularity [5] as its optimization objective. The algorithm is multi-phase, multi-iterative, where within each phase there are multiple iterations, as summarized in Algorithm 1. Initially, each vertex is assigned its own distinct community. Within each iteration, each vertex makes a greedy decision on whether to stay in its current community or migrate to a neighboring community as dictated by the modularity gain. Modularity is calculated after every iteration based on the current state of communities, and a phase terminates when there is "negligible" gain in overall modularity between consecutive iterations (as determined by a threshold τ). At the end of a phase, the graph is compacted such that each community is condensed into one "meta-vertex" and edges are redrawn between those respective meta-vertices as per the connections between the corresponding communities in G. The compacted graph is passed on as input to the next phase. The overall algorithm terminates when the net gain in modularity falls below a certain threshold.

To enable parallel processing on distributed memory, the input graph is initially partitioned (in a trivial manner) such that all processes approximately receive the same number of

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Algorithm 1 Parallel Louvain Algorithm (at rank i).
Input: Local portion $G_i = (V_i, E_i)$ of the graph $G = (V, E)$
Input: Threshold τ (default: 10^{-6}).
Notation: C denotes communities, and Q denotes modularity.

```
$C_{curr} \leftarrow \{\{u\} | \forall u \in V\}$
$\{Q_{curr}, Q_{prev}\} \leftarrow 0$
while true do
    $Q_{curr} \leftarrow LouvainIteration(G_i, C_{curr})$
    if $Q_{curr} - Q_{prev} \leq \tau$ then
        break and output the final set of communities
    $NextPhase(G_i, C_{curr})$
    $Q_{prev} \leftarrow Q_{curr}$
```

vertices. Edges are also partitioned. There are two types of edges—those that connect two vertices that reside locally, and those that connect a local vertex to a "ghost" vertex that resides remotely on another process. Consequently, each process maintains two lists, one for its ghost vertices and another for its ghost communities (along with their owning process ids). The mapping of vertices to processes changes after every phase (owing to graph compaction), and we perform a single (one-time per phase) send-receive communication step to exchange these ghost coordinate information. On the other hand, the community compositions could change more frequently as vertices could change community affiliations at every iteration, and therefore, membership information needs to be relayed from the corresponding owner processes to all those processes that keep a ghost copy of those communities. We also parallelize the graph compaction step, which is a nontrivial step. More details on how these individual steps are implemented in distributed memory can be found in [3].

B. Heuristics for performance optimization

On top of our baseline implementation, we also implemented three different heuristics aimed at improving the overall execution time and/or exploiting quality-time tradeoffs that are exposed through certain algorithmic properties. These heuristics are summarized below.

a) Threshold Cycling: In the baseline algorithm, the threshold parameter τ used to detect phase termination, is kept fixed throughout the execution. However this parameter can be tuned across phases to potentially accelerate convergence. Intuitively, during the initial phases when the graph is relatively large, the threshold is also kept large, to incentivize graph compaction early on. As the algorithm proceeds to its later phases, the threshold can be reduced to make the algorithm sensitive to quality (minor net gains in modularity). This could potentially result in faster convergence of the algorithm, although with varying impact on quality. There are a couple of different variants possible for this idea, and in this paper, we used threshold cycling, in which the threshold is modulated in a cyclical fashion across phases. A range of threshold values are invoked in successive phases after every N phases, where N is predetermined.

b) Early Termination: In our parallel algorithm, one of the major contributors to communication cost is exchange of ghost vertex information across processes. After experimenting with our multithreaded algorithm [6] with numerous inputs, we made the critical observation that the rate at which modularity increases significantly decreases as iterations advance within a phase. This *diminishing returns* property in quality happens because the rate at which vertices change their community affiliation tends to drastically decrease as the iterations progress.

To exploit this expected behavior, we devised a probabilistic scheme that we call "early termination" (*et*), where a vertex decides to stay "active" or get "terminated", at any given iteration. A vertex that is "active" at any iteration reevaluates its community state; whereas a terminated vertex does *not* perform any computation or generate any communication. To identify which vertices to terminate, we look at the most recent activity of that vertex. Intuitively, if it has not moved recently then we reduce the probability that it will stay active. For example, consider vertex v. Let $C_{v,j}$ denote the community containing v at the end of iteration j. Let the probability that v is *active* during iteration k be denoted by $P_{v,k}$. We define $P_{v,k}$ as follows [3]:

$$P_{v,k} = \begin{cases} P_{v,k-1} * (1 - \alpha), & \text{if } C_{v,k-1} = C_{v,k-2} \\ 1, & \text{otherwise} \end{cases} \qquad (1)$$

where α is a real number between 0 and 1. As α approaches zero, it becomes similar to the baseline scheme; and as it approaches one, it becomes more aggressive in terminating vertices early on. We developed two minor variants of early termination, labeled *et* and *etc*. In both *et* and *etc*, when the probability for a given vertex becomes less than some small percentage (2% in our experiments), we terminate it. Furthermore, in *etc*, if the fraction of terminated vertices reaches a percentage (90% in our experiments), we terminate the phase. The latter is implemented using global communication in *Louvain* iteration.

c) Incomplete Coloring: In the serial algorithm, the order in which vertices are processed within each iteration could impact performance or output quality. In parallel, the order could play a prominent role in performance as concurrent processing of two vertices that are connected by an edge (dependency) could delay convergence. In our multithreaded implementation [6], we used distance-1 coloring to overcome this challenge. In a distance-1 coloring, two vertices that are connected by an edge are assigned to two different color classes. We therefore allow concurrent processing of one color class at a time, since this guarantees that no two neighboring vertices are processed concurrently.

In our distributed memory implementation, we perform an incomplete coloring to reduce the overhead in switching between the colors. The basic idea is to color only a fraction of the vertices with a preselected number of color classes using the Jones-Plassmann algorithm [7]. The algorithm proceeds by assigning a unique random number to each vertex. At a given iteration of the algorithm, if the random number of a vertex is the minimum (alternatively, maximum) among its neighbors, then the vertex colors itself at this step with a predetermined color class for that iteration and removes itself from further consideration. Otherwise, it competes in subsequent steps until it gets a color or the given number of colors are exhausted. A second variant of this approach is to keep coloring until a certain minimum fraction of the vertices are colored, after which the remaining vertices are bundled into one color. Note that the vertices thus bundled into the final color class may have conflicts.

III. Experimental Results

We perform our primary experimental evaluations on NERSC Edison, which is a 5,586-node Cray XC30 machine with dual-socket 12-core Intel® "Ivy Bridge" Xeon® E5-2695v2 processor at 2.4 GHz (24 cores per node), 64 GB memory per node, 30 MB L3 cache, and Cray XC series interconnect (Aries) with dragonfly topology. We used Cray MPICH 7.6.2 for this machine. `Vite` was run using 12 MPI processes per node and 2 OpenMP threads per process. We used the Intel® ICPC 17.0.4 compiler with "-O3 -xHost" as compilation options for building Vite. Below, we summarize the descriptors/legends used in the figures/tables in this section to refer to the different variants of our parallel algorithm (discussed in §II).

- `baseline`: the basic parallel version (Algorithm 1) without the heuristics;
- `tscale`: version with threshold cycling enabled;
- `et`: version with adaptive early termination, which requires an input parameter (α). We report *et* performance with $\alpha = 0.25$ (denoted as $et1$) and $\alpha = 0.75$ (denoted as $et2$);
- `etc`: variant of early termination with an extra communication step to gather inactive vertex count. We report etc performance with $\alpha = 0.25$ (denoted as $etc1$) and $\alpha = 0.75$ (denoted as $etc2$); and
- `color`: version with incomplete coloring. We use from 32 to 40 colors in our experiments.

The number of vertices and edges in the 2018 official streaming partition challenge datasets are listed in Table I.

TABLE I
Official static dataset characteristics.

Input label	#Nodes	#Edges
1K	1,000	8,067
5K	5,000	50,850
20K	20,000	473,914
50K	50,000	1,189,382
200K	200,000	4,750,333
1M	1,000,000	23,716,108
5M	5,000,000	118,738,395
20M	20,000,000	475,167,612

We begin with quality comparisons using the ground truth information for a subset of the inputs. We then provide detailed information on several aspects of performance by providing strong scaling results, comparison of different heuristics on quality-performance tradeoffs, and a detailed summary of best performance (Table IV). We also provide relevant observations and analysis within each subsection.

A. Qualitative Comparisons

In order to assess the quality of `Vite`, we compare our results against reference ground truth files for small-medium sized datasets, as shown in Table II. We compute precision, recall and F-score using the formulas detailed in [8]. When the quality assessment feature is turned on, `Vite` performs extra collective operations per phase to gather the vertex-to-community mapping of the current graph into the master process (i.e., rank #0).

We note that, for the cases with the very low F-Score values (e.g., 5K HIHI and HILO, 20K HIHI, 50K HIHI, most of 200K), we have identified potential issues in the current ground truth partitions and input files provided in the challenge dataset, and therefore, do not reflect the true quality of the solutions computed by `Vite`. For example, some of the inputs have isolated vertices (i.e., some vertex ids do not appear in the edge list). Isolated vertices should be partitioned in their own cluster, but in the ground truth information they are assigned to partitions with other vertices.

B. Performance

We now evaluate the performance of `Vite` on both the challenge networks and other publicly available real world and synthetic datasets. We recently demonstrated speedups between $2 - 46\times$ over the baseline version using different heuristics for a large set of inputs [3]. However, the efficacy of these heuristics depends on the connectivity structure of the inputs. As an illustration, we show the scaling of `Vite` on four Graph500 [9] Kronecker graphs that have a poor community structure (modularity$\approx 0.0107 - 0.0199$) in Figure 1. While the figure shows strong scaling on up to 1536 processes that `Vite` is able to achieve on these graphs (about 2.3-3.5x), it also shows the negligible difference that heuristics make with respect to the baseline performance.

Next, we show the results obtained on four of the largest challenge networks (200K, 1M, 5M, 20M) under the four different categories of hardness (LOLO, LOHI, HIHI, HILO). Figure 2 and Figure 4 shows the performance, and Figure 3 shows the quality, using different heuristic settings for `Vite`. These two figures illustrate the following: The coloring version is the slowest (on average about $8\times$) compared to all other heuristic versions, as shown in Figure 4. However, it is also the one that consistently delivers the highest modularity (as shown in Figure 3). Furthermore, Figure 3 also shows the impact of the input hardness on the heuristics. Specifically, the faster heuristics (i.e., without coloring) generally perform significantly better in quality as well on the less harder cases (e.g., LOLO, LOHI) than for the hardest case (HIHI), as shown on Figure 2. The coloring heuristic require extra communication to synchronize after each round of processing vertices with the same color, which increases the overall execution time relative to the baseline version. The extra communication step in *etc* can help in avoiding reasonable computation/ communication overhead associated with "ghost" vertices. Although a reduction in the number of iterations per phase may improve overall performance (we observe about $1.25 - 2.3\times$ improvement in *etc* as compared to *et* for certain cases); but on some other cases, this effect may lead to an increased number of phases, due to a large number of inactive vertices affecting the general convergence. Also, if the number of iterations per phase is relatively small originally (as observed for some LOHI and HIHI inputs), then the communication overhead in *etc* may invalidate any benefit over *et*. Figures 2 and 3 serve to demonstrate the performance-quality tradeoffs that can be achieved through different heuristics.

a) Combining heuristics delivers better performance: Some of the heuristics can be combined with others to generate a complementary positive effect (on performance

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

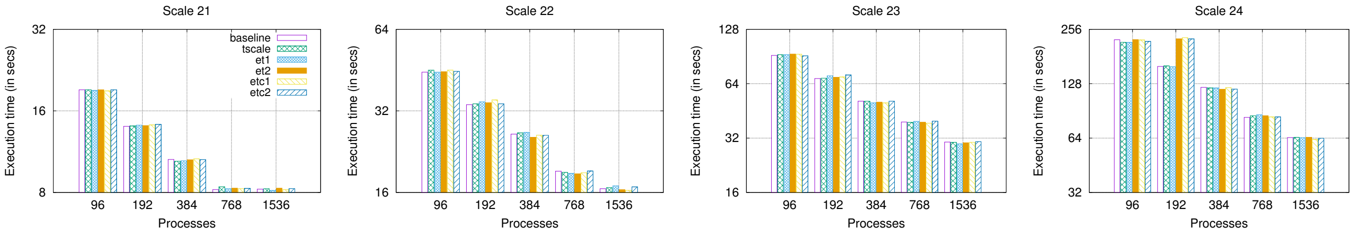

Fig. 1. Parallel heuristics have little effect on RMAT generated Graph500 graphs.

TABLE II
QUALITY COMPARISONS OF STATIC DATASETS (1K-200K) WITH KNOWN GROUND TRUTH COMMUNITY INFORMATION. WE LIST PRECISION, RECALL AND F-SCORE FOR EACH TYPE OF INPUT. VERY LOW F-SCORE VALUES ARE DUE TO ISSUES IN THE GROUND TRUTH FILES FROM CHALLENGE DATASET.

$\|V\|$	1,000			5,000			20,000			50,000			200,000		
Input	**Prec.**	**Rec.**	**F-Sc.**	**Prec.**	**Rec.**	**F-Sc.**	**Prec.**	**Rec.**	**F-Sc.**	**Prec.**	**Rec.**	**F-Sc.**	**Prec.**	**Rec.**	**F-Sc.**
Baseline															
LOLO	0.966	0.988	0.976	0.685	0.976	0.805	1	1	1	0.938	0.993	0.965	0.956	1	0.977
LOHI	0.461	0.819	0.590	0.527	0.874	0.658	0.921	0.997	0.957	0.8700	0.998	0.929	0.0220	0.0399	0.0283
HIHI	0.660	0.911	0.765	0.201	0.00475	0.00928	0.858	0.00346	0.00690	0.0377	0.0613	0.0467	0.0191	0.0332	0.0243
HILO	0.580	0.939	0.717	0.0636	0.0769	0.0696	1	1	1	0.983	1	0.991	0.0156	0.0287	0.0202
Baseline + Color															
LOLO	0.717	0.951	0.818	0.871	0.979	0.922	0.943	0.994	0.968	0.950	0.986	0.968	0.786	1	0.880
LOHI	0.793	0.945	0.862	0.595	0.921	0.723	0.0591	0.0721	0.0649	0.838	0.999	0.911	0.317	0.991	0.481
HIHI	0.643	0.907	0.752	0.0990	0.189	0.130	0.475	0.991	0.643	0.389	0.948	0.552	0.338	0.998	0.505
HILO	0.824	0.927	0.872	0.911	0.995	0.951	0.907	0.976	0.940	0.914	1	0.955	0.466	1	0.635

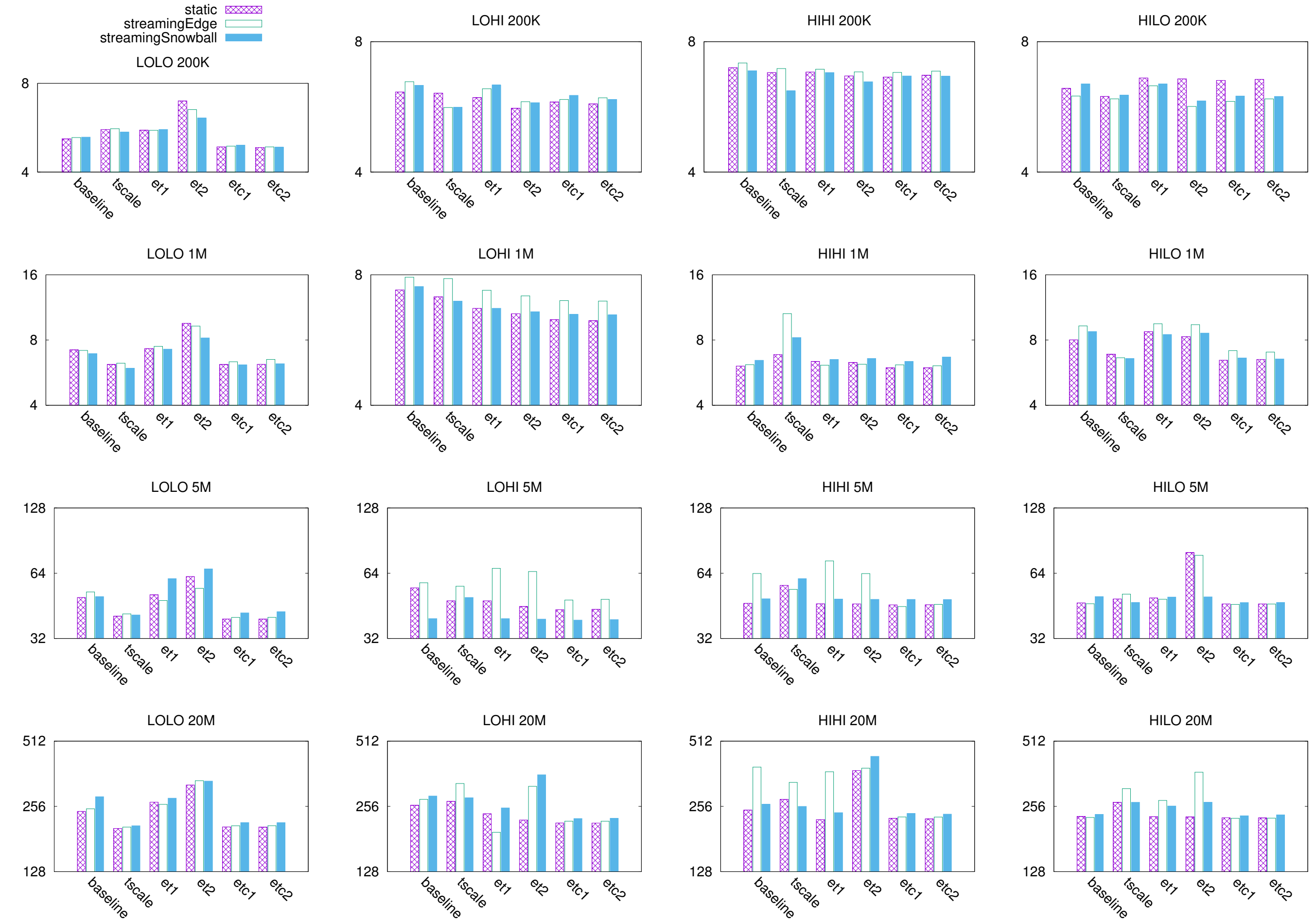

Fig. 2. Runtime performance (*Y-axis*, in secs.) for four of the largest official 2018 datasets. Graphs of sizes 1M, 5M and 20M are executed on 192 processes (16 nodes), whereas the 200K graph is executed on 24 processes (1 node) of NERSC Edison.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

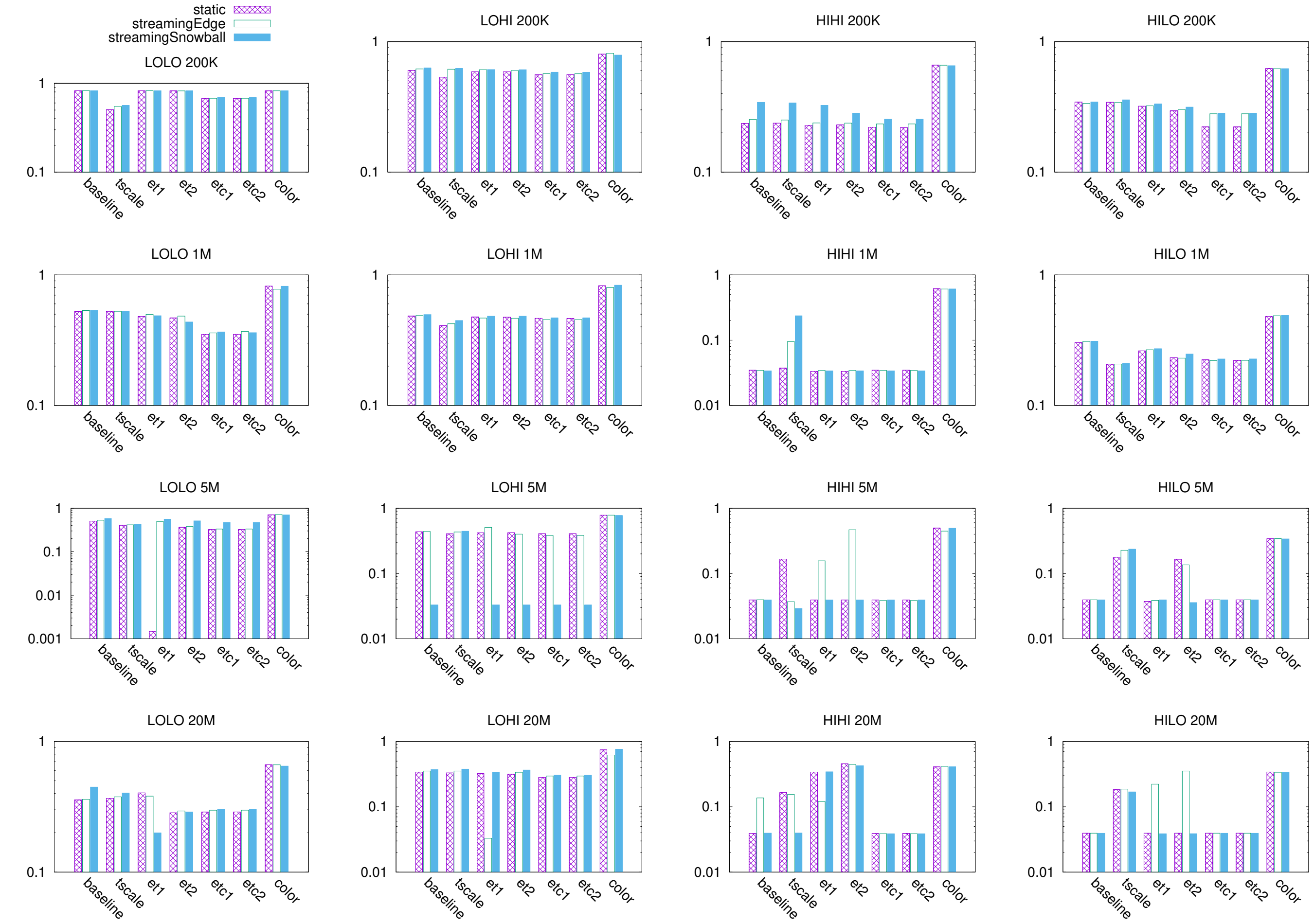

Fig. 3. Modularity (*Y-axis*) of official 2018 datasets of sizes 200K, 1M, 5M and 20M. Graphs of sizes 1M, 5M and 20M are executed on 192 processes (16 nodes), whereas the 200K graph is executed on 24 processes (1 node) of NERSC Edison.

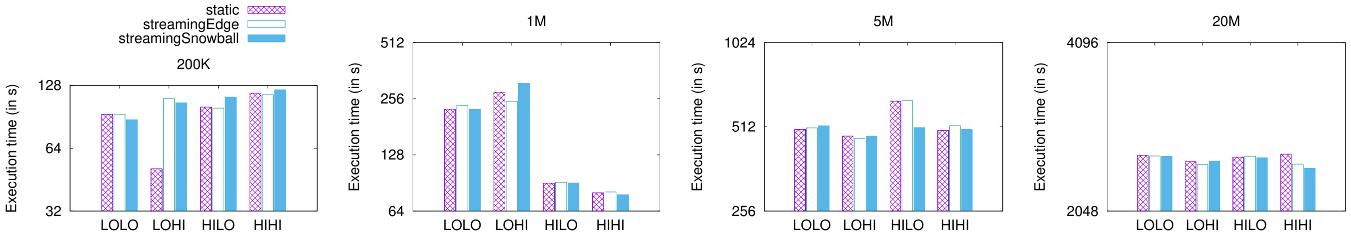

Fig. 4. Runtime performance for four of the largest official 2018 datasets with coloring heuristic. Graphs of sizes 1M, 5M and 20M are executed on 192 processes (16 nodes), whereas the 200K graph is executed on 24 processes (1 node) of NERSC Edison.

and/or quality). As a test, we combined early termination (*et*) with coloring. Figures 8 and 9 show this effect of combining heuristics for two input cases (respectively): LOHI and Orkut. We can observe that coloring combined with one of the more aggressive versions of early termination (et2 or etc2), significantly helps in reducing the execution time (in many cases by about 10×) without lowering quality. Improvement in the execution time originates from the fact that coloring helps in generating an informed partial ordering (i.e., processing of vertices). This allows vertices to settle in their final community states more quickly. Early termination exploits this behavior and quickens convergence (by reducing the number of iterations), as confirmed by the data presented in Table III.

TABLE III
NUMBER OF ITERATIONS (AND PHASES) FOR LOHI STATIC DATASETS USING COLORING COMBINED WITH *et*.

Heuristics /Sizes	**1M**		**5M**		**20M**	
	Iters.	**Phases**	**Iters.**	**Phases**	**Iters.**	**Phases**
color	175	4	163	4	273	5
color+et1	182	4	202	6	139	4
color+et2	147	4	148	4	108	5
color+etc1	70	5	115	7	92	6
color+etc2	72	5	92	6	92	6

b) Scaling results on other inputs: We demonstrate the scalability of Vite using the largest reference dataset, 20M (Figure 5), as well as other datasets, such as the protein k-

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

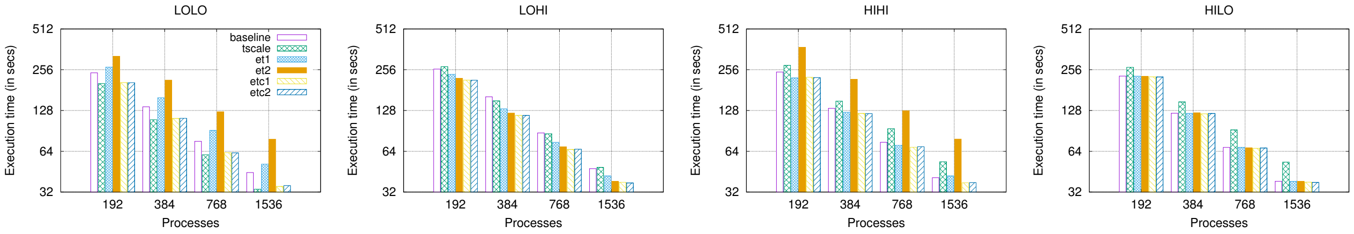

Fig. 5. Scalability of different variants on the largest reference static dataset, 20M.

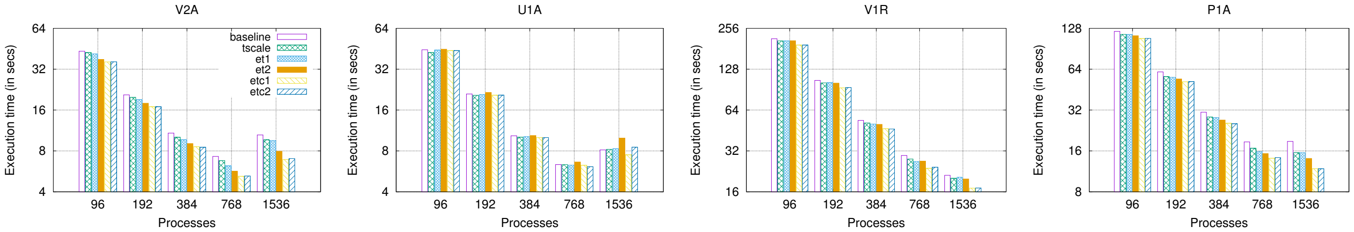

Fig. 6. Scalability of protein k-mer graphs.

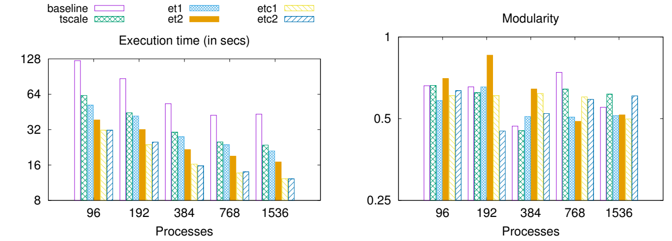

Fig. 7. Scalability and modularity variation of Orkut (3M nodes, 117M edges).

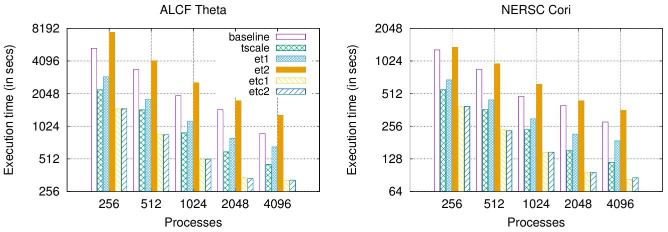

Fig. 10. Scalability of Friendster (65.6M nodes, 1.8B edges) on NERSC Cori (Haswell) and ALCF Theta (KNL).

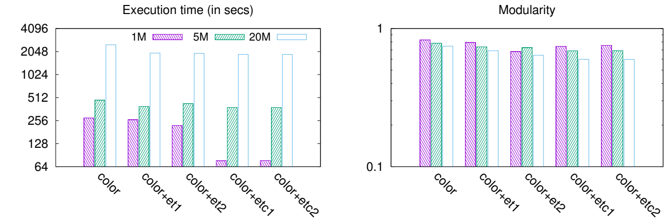

Fig. 8. Performance on LOHI static datasets (1M, 5M, 20M) using 192 processes when coloring is combined with *et*.

mer graphs (Figure 6) and social network graphs, namely Friendster (Figure 10) and Orkut (Figure 7).

c) Performance scaling on other platforms: We have tested Vite on multiple platforms. Figure 10 shows results of Vite for Friendster dataset on NERSC Cori and ALCF Theta platforms. We use the recommended quad-cache mode for the KNL nodes in Theta (without changing our code). We observe that the Intel® Haswell® nodes of NERSC Cori outperform Theta by a factor of 2 to 3×.

d) Best heuristics for input datasets: In conclusion, we summarize the execution times for a selected set of different inputs and configurations in Table IV. All the datasets used in our evaluation are listed on the GraphChallenge website (http://graphchallenge.mit.edu/data-sets).

TABLE IV
A SUMMARY OF BEST PERFORMANCE RESULTS (RUN ON NERSC CORI/EDISON PLATFORM) FOR INPUT GRAPHS.

Graphs	Nodes	Edges	Mod.	Time(in s)	Proc.	Best heuristic
UK2007	105.8M	3.3B	0.972	32.08 (Cori)	1024	*etc2*
Friendster	65.6M	1.8B	0.624	82.72 (Cori)	4096	*etc1*
Web-cc12-PayLevel	42.8M	1.2B	0.687	104.09 (Cori)	1024	*etc1*
Web-wiki-en-2013	27.1M	601M	0.671	16.89 (Cori)	2048	*et2*
LOLO Static	20M	475.1M	0.358	33.6 (Edison)	1536	*tscale*
LOHI Static	20M	475.1M	0.34	37.4 (Edison)	1536	*etc2*
HIHI Static	20M	474.9M	0.0391	37.5 (Edison)	1536	*etc1*
HILO Static	20M	475M	0.0394	37.7 (Edison)	1536	*etc2*
K-mer P1a	139.3M	297.8M	0.971	11.8 (Edison)	1536	*etc2*
K-mer U1a	67.7M	138.8M	0.988	7.5 (Edison)	1536	*etc1*
K-mer V1r	214M	465.4M	0.944	17 (Edison)	1536	*etc1*
K-mer V2a	55M	117.2M	0.990	6.91 (Edison)	1536	*etc1*
Orkut	3M	117.1M	0.661	11.36 (Cori) / 12.19 (Edison)	1024/ 1536	*etc1*

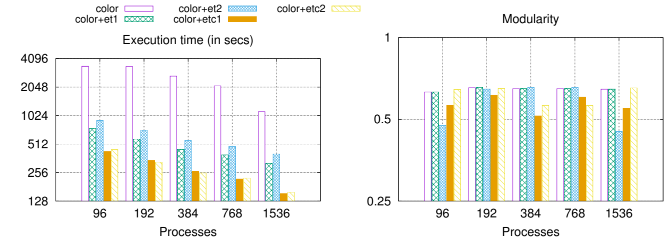

Fig. 9. Orkut performance when coloring is combined with *et*.

Acknowledgements

The research is in part supported by the U.S. DOE ExaGraph project, the DARPA HIVE Program and HPDA at DOE PNNL, the DOE award DE-SC-0006516 to WSU, the NSF award 1815467, and the NSF award IIS 1553528. 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] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, p. Accepted.

[4] 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.

[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] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19–37, 2015.

[7] 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.

[8] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo, "Scalable static and dynamic community detection using grappolo," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–6.

[9] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray UserâĂŹs Group (CUG)*, vol. 19, pp. 45–74, 2010.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE