Scaling and Quality of Modularity Optimization Methods for Graph Clustering

Sayan Ghosh*, Mahantesh Halappanavar*, Antonino Tumeo*, Ananth Kalyanaraman†

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

***Abstract*—Real-world graphs exhibit structures known as "communities" or "clusters" consisting of a group of vertices with relatively high connectivity between them, as compared to the rest of the vertices in the network. Graph clustering or community detection is a fundamental graph operation used to analyze real-world graphs occurring in the areas of computational biology, cybersecurity, electrical grids, etc. Similar to other graph algorithms, owing to irregular memory accesses and inherently sequential nature, current algorithms for community detection are challenging to parallelize. However, in order to analyze large networks, it is important to develop scalable parallel implementations of graph clustering that are capable of exploiting the architectural features of modern supercomputers.**

In response to the 2019 Streaming Graph Challenge, we present quality and performance analysis of our distributed-memory community detection using *Vite*, which is our distributed memory implementation of the popular *Louvain* method, on the ALCF Theta supercomputer.

Clustering methods such as *Louvain* that rely on modularity maximization are known to suffer from the resolution limit problem, preventing identification of clusters of certain sizes. Hence, we also include quality analysis of our shared-memory implementation of the Fast-tracking Resistance method, in comparison with Louvain on the challenge datasets.

Furthermore, we introduce an edge-balanced graph distribution for our distributed memory implementation, that significantly reduces communication, offering up to 80% improvement in the overall execution time. In addition to performance/ quality analysis, we also include details on the power/energy consumption, and memory traffic of the distributed-memory clustering implementation using real-world graphs with over a billion edges.

I. Introduction

Graph clustering, popularly known as community detection, can be broadly defined as the partitioning of the vertex set $V(G)$ of a graph $G = (V, E, \omega)$ into clusters (or partitions) such that the vertices within a cluster share a tightly knit connection among them, while a sparser connection to rest of the network. Community detection has emerged as an important tool to discover and describe important characteristics of a graph with numerous applications in science and analytics [1].

Although several variants of the community detection problem are known to be NP-hard [2], [3], a diverse set of efficient heuristic strategies have been developed. Modularity optimization is an important approach and the Louvain method has been demonstrated to compute good quality solutions efficiently [1]. However, modularity optimization based methods also suffer from the so-called resolution-limit problem, where small clusters tend to get merged with larger clusters, and thus, resulting in mis-identification in graphs that contain both large and small clusters. Heuristics have been developed to address the resolution limit problem [4]. Fast-tracking resistance is one such method that we consider in this work as a baseline for comparison (§II).

In this paper, we present our work on static community detection using modularity optimization methods in response to the 2019 Graph Challenge [5]. The key contributions are two-fold: (i) we present results from executing an optimized implementation of `Vite`, which is our parallel implementation of the Louvain algorithm, for community detection on distributed-memory systems [6]—using the ALCF Theta supercomputer as our experimental testbed (details provided in §II-B); and (ii) we also present results of implementing a shared memory method to address the resolution limit problem that modularity optimization methods typically face.

These contributions can be summarized as follows:

- Present qualitative results from parallel Louvain and Fast-Tracking Resistance for the Graph Challenge dataset (§III);
- Present strong scaling results and miscellaneous analysis on ALCF Theta for a set of inputs consisting of synthetic and real-world instances (§III); and
- Present relative comparison with the Graph Challenge reference implementation for the static community detection inputs.

II. Methodology

A variety of community detection techniques has been proposed in literature. Methods that optimize modularity have been demonstrated to not only scale but also to compute good quality solutions, albeit with known limitations [1]. We provide a brief background on modularity-based graph clustering and our target hardware platform in this section.

A. Modularity-based Graph Clustering

Modularity is a statistical measure to assess the quality of a given community-wise partitioning of a graph. Given a partitioning, $P = \{C_1, C_2, \ldots C_k\}$ of the vertex set V in $G(V, E, \omega)$, where $1 \leq k \leq n$, *modularity* Q is given by $X - Y$, where X is the fraction of intra-cluster edges imposed by the partitioning P on V, and Y is the expected fraction in an equivalent but randomly reconnected graph with the same number of vertices, edges and vertex degree distribution [7].

The *Louvain algorithm* proposed by Blondel *et al.,* begins by assigning a unique community to each vertex and repeatedly seeking new assignments for a vertex (current assignment

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

or one of the communities of its neighbors) that maximizes the gain in modularity for that vertex [8]. Within a given iteration, new assignments are sought for every vertex considered in an arbitrary order. The algorithm iterates until modularity gain is above a given threshold value. Further, the algorithm proceeds to the next phase by collapsing the vertices belonging to a community to a single (meta) vertex in the next phase, and adding edges to account for inter and intra-community (self loop) edges. The operations are repeated until the coarsened graph is of a certain size.

The Louvain algorithm as proposed is inherently sequential, and several challenges arise in parallelizing the algorithm. Lu *et al.* proposed several heuristics for shared-memory platforms [9], and Ghosh *et al.* adapted some of these heuristics and developed new ones for distributed-memory platforms [10]. We use `Vite` developed by Ghosh *et al.* in this work and adapt it for the Intel Knights Landing processor.

Methods based on modularity optimization have been shown to merge smaller clusters with neighboring larger clusters, known as the *resolution limit problem* [11]. Addressing resolution limit in an efficient manner is difficult, and many of the methods proposed also suffer the same problem [4]. We developed a shared-memory implementation of the *fast-tracking resistance* (FTR) method proposed by Granell *et al.* [4]. FTR begins from a partition (clustering) and attempts to split the larger partitions into smaller ones using a parameter called resistance. We present results obtained from FTR as a baseline for comparison with parallel Louvain (§III).

B. Hardware Platform

Throughout the last decade, we have experienced an increase in the core count (approximately 10-50×) of processors in HPC systems. HPC systems have also significantly increased their heterogeneity, by integrating loosely coupled workload-specialized throughput processors (i.e., general purpose graphic processing units) or tightly coupled extended vector units (512-bit and beyond). This has made arithmetic operations (and, in particular, floating point operations) cheap in terms of energy and actual costs. However, network and memory bandwidth are not increasing at the same rate, providing ratios ranging from 10 to 100 flops per bytes and resulting in unbalanced systems, especially for the novel memory bound workloads of data analysis and machine learning.

A key challenge for increasing bandwidth with conventional DDR memory lies in the additional pins, and energy, required to drive the memory chips and the related memory channels. 3D-stacked memory allows stacking multiple DRAM (Dynamic Random Access Memory) chips one on top of the other, and interconnecting them to a logic layer (memory controller) with through silicon vias (TSVs), providing high bandwidth with low energy costs. HBM (High Bandwidth Memory), the standard ratified by the JEDEC, where the design of the logic layer is left to the processor integrator, is by far the predominant type of 3D stacked memory, for a period opposed to Micron HMC (Hybrid Memory Cube). A number of CPUs (e.g., Intel Xeon Phi "Knights Landing" -KNL-, Fujitsu A64FX), GPUs (e.g., NVIDIA Pascal and Volta, AMD Radeon Vega) and Vector engines (e.g., NEC SX Aurora) used in modern HPC systems have started to integrate this type of memory, leading to interesting trade-offs in terms of bandwidth and memory density (currently stacks only up to 32 GB are possible).

In this work we use ALCF Theta for performance evaluation. Theta is a 4,392 nodes Cray system with Cray Aries interconnect (with Dragonfly topology). Each node contains an Intel KNL processor with 64 cores at 1.3 Ghz (Intel Silvermont architecture based cores), 16 GB of high-bandwidth in-package memory (named MCDRAM), 192 GB of DDR4 RAM, and a 128 GB SSD.

Theta represent a stepping stone towards the next generation Aurora supercomputer, claimed to become (when operational in 2021) the first US DOE exascale system. While Aurora will be based on a heterogeneous design including Intel Xeon and Intel Xe architecture based GPUs, KNL still remain an interesting platform for evaluation of novel workloads, due to its use of many simple cores, the large vector units (Intel AVX-512), and the use of MCDRAM, an HBM-based 3D-stacked memory.

These design points, although with very different implementations and solutions, will be used in future processors: Fujitsu A64X will be a ARM manycore design with the Scalable Vector Instruction Extensions and HBM2 memory (with a more balanced flop/byte ratio). The KNL design has, for example, exposed some key aspects that developers need to take into account for the effective exploitation of MCDRAM (for e.g., higher performances if different processes are used to map on the cores that directly interface with a vault - i.e., a 3D bank of the stacked memory, or limited effectiveness when used in software cache mode for the larger DDR4 memory of the system) [12] and of its multithreaded processors.

III. Evaluation

We perform our experimental evaluations on the ALCF Theta supercomputer, and we discuss the platform in Section II-B. A KNL node in Theta consists of 64 cores, organized into 32 tiles (2 cores/tile, sharing an L2 cache of 1 MB) in a 2-D layout, a high bandwidth on-package memory of size 16 GB (MCDRAM), and 192 GB of DDR4 main memory. The tiles are connected by a mesh interconnect, and the mesh support different levels of memory address affinities, known as clustering modes. We use the *quadrant* clustering mode, in which the tiles are divided into four parts (quadrants), which are spatially located near four groups of memory controllers.

It is possible to configure the MCDRAM as a cache for the main memory (*cache* mode), and also treat it as addressable memory (*flat* mode). The access latency of MCDRAM is higher than the standard CPU caches, which are usually multi-way set associative (reducing conflict misses), whereas MCDRAM in cache mode is direct-mapped. Due to the irregular memory accesses inherent in graph applications, we observed (up to 30%) better performance for our Louvain implementation using the KNL MCDRAM in flat mode, as compared to the aforementioned cache mode. Therefore, we use the MCDRAM in flat mode for the current evaluations. We use a custom memory allocator (i.e., `hbw::allocator`)

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

from the `memkind` library [13] to allocate some of the heavily used C++ data structures on the MCDRAM.

We used Cray MPICH 7.7.3 as the MPI implementation for this machine. Our distributed-memory Louvain implementation was run using 16 MPI processes per node and 4 OpenMP threads per process. We used the Intel® ICPC 18.0.1 compiler with "-O3 -xmic-AVX512" as compilation options. We also use Cray Performance Analysis Tool (CrayPat) to report power and memory usage metrics.

In the rest of this section, we discuss the quality/ performance of our graph clustering implementations and compare them with the official 2019 challenge datasets. We postulate that massive scale dynamic clustering would require efficient data structures, and usage of persistent memory, which are still not available on current HPC systems. At present, our implementation is lacking support of clustering on streaming/dynamic graph data. Therefore, for the streaming cases we concatenate the individual files and report clustering results based on a single file.

Apart from the challenge datasets, we perform clustering analysis on eight real-world networks (publicly available from the Suitesparse Matrix Collection [14]), all of which have over a billion edges. In Table I, we list the size of the undirected representations of the input networks (challenge and real-world) used in the evaluations. The Graph Challenge streaming partition datasets are denoted as LOLO, LOHI, HILO and HIHI. These acronyms (to be read as low/high) represent the level of overlap and relative size variation between the blocks. For instance, HIHI denotes "high level of overlap and a high level of size variation between blocks", indicating stronger interactions (i.e., a large number of edges or connections) between the individual blocks or clusters of different sizes, making the task of community detection harder.

TABLE I
EVALUATION DATASETS

Graph challenge datasets			Real-world datasets		
Name	**#Vertices**	**#Edges**	**Name**	**#Vertices**	**#Edges**
lowOverlap-lowBlockSize (LOLO),	1K	16K	mycielskian20	786.4K	1.35B
	5K	102.3K	webbase-2001	118.1M	2.03B
lowOverlap-highBlockSize (LOHI),	20K	946.6K	it-2004	41.2M	2.3B
	50K	2.37M	twitter7	41.6M	2.93B
highOverlap-highBlockSize (HIHI),	200K	9.5M	MOLIERE_2016	30.2M	3.33B
	1M	47.5M	com-Friendster	65.6M	3.61B
highOverlap-lowBlockSize (LOHI)	5M	237.5M	sk-2005	50.6M	3.89B
	20M	949.9M	uk-2007	105.8M	6.6B

A. Analysis of Graph Challenge official datasets

We ran the baseline (serial) partition challenge Python code [5] for small datasets and compare the quality with our clustering implementations, as shown in Table II. The quality metrics of precision, recall and F-score are computed by comparing the results of a clustering implementation with the provided ground truth data. High values of F-score (close to 1) and precision/recall corresponds to high quality solution as compared to ground truth community assignments. For our distributed-memory Louvain implementation, we observe lower F-scores for the static HIHI cases, as compared to baseline (for e.g., static datasets with $5,000$ vertices). For few of these cases, the Fast-Tracking Resistance counterpart provides a better solution, which is nearly identical to the baseline scores.

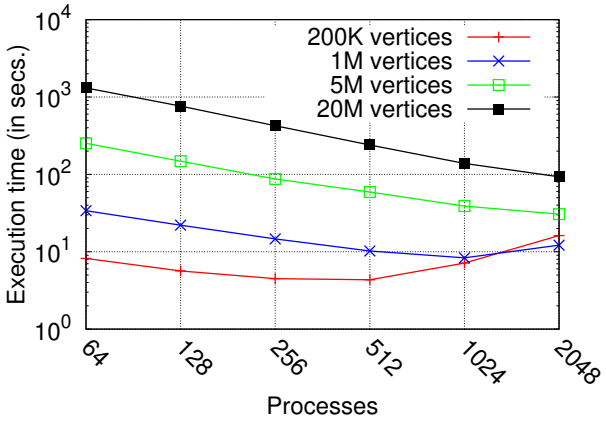

Fig. 1. Distributed Louvain clustering strong scaling results of highOverlap-highBlockSize challenge datasets on ALCF Theta.

In some cases, our clustering scores are notably low due to a significant increase in "false positive" scenarios (thereby lowering the *precision*) compared to the "true positive" ones, i.e., when a pair of vertices belong to the same community in the current cluster output set, but differs in the ground truth set. This increase in "false positive" scenarios can be attributed to the presence of a number of isolated vertices or islands in certain challenge datasets, which are grouped in distinct communities as per our design, whereas in the challenge ground truth data, they are merged into a larger community.

Apart from quality metrics, we demonstrate strong scaling of our distributed-memory Louvain implementation using the larger challenge datasets in Figure 1. We observe a speedup of 2-14$\times$ on 2048 processes of ALCF Theta.

B. Analysis of real-world graphs

We chose eight large real-world graphs (listed in Table I) to analyze the performance of our distributed clustering implementation. Load balancing of real-world graphs is challenging, since it is nontrivial to implement equitable partitioning of graphs across processes.

We introduce an edge-balanced partitioning scheme that vastly improves the communication time at the expense of an extra communication step involving a broadcast operation. First, we convert an input graph from its native format to a binary format, embedding the count of edges connected to a vertex, in addition to the edge list information. The binary format is designed to make the I/O efficient, albeit at the expense of a one-time-only file conversion overhead. The binary format also allows avoiding the string parsing overhead of reading ASCII text files, and uses relatively less storage. We use MPI collective I/O to read the binary data into the graph CSR data structure directly, and in our experiments, the file I/O part usually took 1-2% of the entire execution time.

Our assumption is that the number of edges in a graph is much more compared to the number of vertices, and reading

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

TABLE II

QUALITY COMPARISONS OF OFFICIAL DATASETS (1K-1M) WITH KNOWN GROUND TRUTH COMMUNITY INFORMATION (USING LOUVAIN FOR DISTRIBUTED-MEMORY AND FAST TRACKING RESISTANCE FOR SHARED-MEMORY, COMPARED WITH SERIAL BASELINE BLOCKMODEL PARTITIONING IMPLEMENTATION). ENTRIES MARKED AS '-' WERE TOO SMALL, AND ENTRIES MARKED AS 'X' WERE NOT COMPUTED.

Baseline blockmodel partitioning implementation (in Python) results of STATIC datasets																		
Input	**1000**			**5000**			**20000**			**50000**			**200000**			**1000000**		
	Prec.	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**
LOLO	0.998	0.998	0.998	0.921	1.000	0.959	1.000	1.000	1.000	x	x	x	x	x	x	x	x	x
LOHI	0.919	0.740	0.820	0.935	0.874	0.903	0.840	0.456	0.591	x	x	x	x	x	x	x	x	x
HILO	0.868	0.969	0.915	0.844	0.765	0.802	0.937	1.000	0.967	x	x	x	x	x	x	x	x	x
HIHI	0.816	0.851	0.833	0.635	0.680	0.657	0.538	0.673	0.598	x	x	x	x	x	x	x	x	x
Distributed-memory Louvain clustering results of STATIC datasets																		
Input	**1000**			**5000**			**20000**			**50000**			**200000**			**1000000**		
	Prec.	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**
LOLO	0.989	0.984	0.986	0.337	0.976	0.501	1.000	1.000	1.000	1.000	1.000	1.000	0.956	1.000	0.978	0.009	0.015	0.011
LOHI	0.499	0.823	0.621	0.500	0.863	0.634	0.741	0.997	0.851	0.771	0.998	0.870	0.022	0.050	0.031	0.012	0.025	0.016
HILO	0.810	0.928	0.865	0.065	0.073	0.069	1.000	1.000	1.000	0.983	1.000	0.991	0.016	0.027	0.020	0.009	0.027	0.013
HIHI	0.363	0.849	0.509	0.100	0.146	0.118	0.858	0.003	0.007	0.038	0.061	0.047	0.019	0.035	0.025	0.081	-	-
Shared-memory Fast Tracking Resistance clustering results of STATIC datasets																		
Input	**1000**			**5000**			**20000**			**50000**			**200000**			**1000000**		
	Prec.	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**
LOLO	0.989	0.984	0.986	0.989	0.959	0.974	1.000	1.000	1.000	1.000	1.000	1.000	0.956	1.000	0.978	0.455	-	-
LOHI	0.905	0.910	0.908	0.964	0.850	0.903	0.976	0.997	0.987	0.866	0.999	0.928	0.684	0.001	0.001	0.647	-	-
HILO	0.840	0.929	0.882	0.670	0.006	0.012	1.000	1.000	1.000	0.983	1.000	0.991	0.262	-	0.001	0.340	-	-
HIHI	0.783	0.793	0.788	0.566	0.003	0.005	0.812	0.998	0.896	0.546	0.999	0.706	0.324	-	0.001	0.204	-	-
Distributed-memory Louvain clustering results of STREAMINGEDGE datasets (appended invidual files)																		
Input	**1000**			**5000**			**20000**			**50000**			**200000**			**1000000**		
	Prec.	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**
LOLO	0.100	0.100	0.100	0.054	0.055	0.055	0.034	0.035	0.034	0.025	0.025	0.025	0.015	0.016	0.016	0.009	0.017	0.012
LOHI	0.144	0.193	0.165	0.072	0.211	0.107	0.059	0.062	0.060	0.033	0.038	0.035	0.022	0.034	0.027	0.012	0.024	0.016
HILO	0.102	0.203	0.135	0.059	0.122	0.079	0.035	0.036	0.035	0.024	0.025	0.025	0.016	0.022	0.018	0.009	0.027	0.013
HIHI	0.126	0.158	0.140	0.101	0.002	0.003	0.053	0.000	0.000	0.037	0.060	0.046	0.019	0.033	0.024	0.013	-	-
Distributed-memory Louvain clustering results of STREAMINGSNOWBALL datasets (appended invidual files)																		
Input	**1000**			**5000**			**20000**			**50000**			**200000**			**1000000**		
	Prec.	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**	**Prec.**	**Recl.**	**F-scr.**
LOLO	0.100	0.319	0.152	0.054	0.053	0.054	0.034	0.040	0.037	0.025	0.025	0.025	0.015	0.016	0.016	0.009	0.023	0.013
LOHI	0.135	0.004	0.008	0.072	0.074	0.073	0.058	0.005	0.008	0.033	0.038	0.035	0.022	0.035	0.027	0.012	0.025	0.016
HILO	0.104	0.113	0.108	0.059	0.088	0.070	0.035	0.036	0.036	0.024	0.025	0.025	0.015	0.024	0.019	0.009	0.032	0.014
HIHI	0.127	0.264	0.171	0.096	0.002	0.003	0.053	0.000	0.000	0.037	0.091	0.053	0.019	0.040	0.026	0.013	-	-

the edge counts associated with the vertices take only a fraction of the time taken to read the entire graph. Therefore, a single root process can read the graph partially (while rest of the processes are waiting on a barrier), to construct an equitable distribution scheme which is still vertex-based, and, broadcast it to the rest of the processes. In this scheme, processes may own a varying number of vertices (and all the edges connected to those vertices) to balance out the number of edges owned by the processes. Figure 2 demonstrates the standard deviation of edges owned by a process in a standard vertex-based distribution (where each process owns roughly same number of vertices, i.e., $\frac{|V|}{p}$), as compared to our edge-balanced vertex-based distribution, for the real-world graphs.

The impact of the edge-balanced distribution is observed in Figure 3, which shows up to 80% improvement in the end-to-end execution time of clustering compared to the standard distribution, for most of the real-world graphs. Figure 3 also demonstrates a speedup of up to 6× on 2048 processes.

Apart from execution time performance, we also measure the power/energy consumption, MPI communication volume, and memory traffic of our distributed-memory clustering implementation across 16-128 nodes (each node uses 16 processes, and each process uses 4 threads, engaging all the 64 available cores of KNL) of ALCF Theta.

In Table III, we choose four real-world graphs (using the edge-balanced distribution) to study the miscellaneous performance metrics. We calculate the #Edges by summing the edges over iterations and phases of Louvain clustering. The maximum edge processing rate (#Edges/secs.) obtained from the baseline implementation of parallel Breadth First Search (BFS) of the Graph500 benchmark [15] for a similar system (NERSC Cori KNL nodes) is reported in the latest Jun'19 Graph500 list to be 678.933 GTEPS (Giga-Traversed-Edges-Per-Second) on 512 nodes. In comparison, we observe a maximum of 52 MTEPS (Mega-TEPS) on 128 nodes for our clustering implementation, owing to the significant differences between BFS and Louvain clustering, and the respective volumes of data.

Memory traffic corresponds to the total sum (across all the processes) of memory accesses in GigaBytes. Since KNL has the on-package high bandwidth memory (MCDRAM), we separate DDR memory traffic with HBM or MCDRAM traffic. The HBM traffic is significantly more than the DDR traffic for most of the cases (especially for graphs that would typically run for many clustering iterations, like com-Friendster), which corroborates our active use of KNL MCDRAM. However, in certain cases, with increasing number of processes, memory per process shrinks, reducing the HBM access rate as compared to DDR. The memory traffic rate of com-Friendster is 2-10× more than the other graphs owing to a large number

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

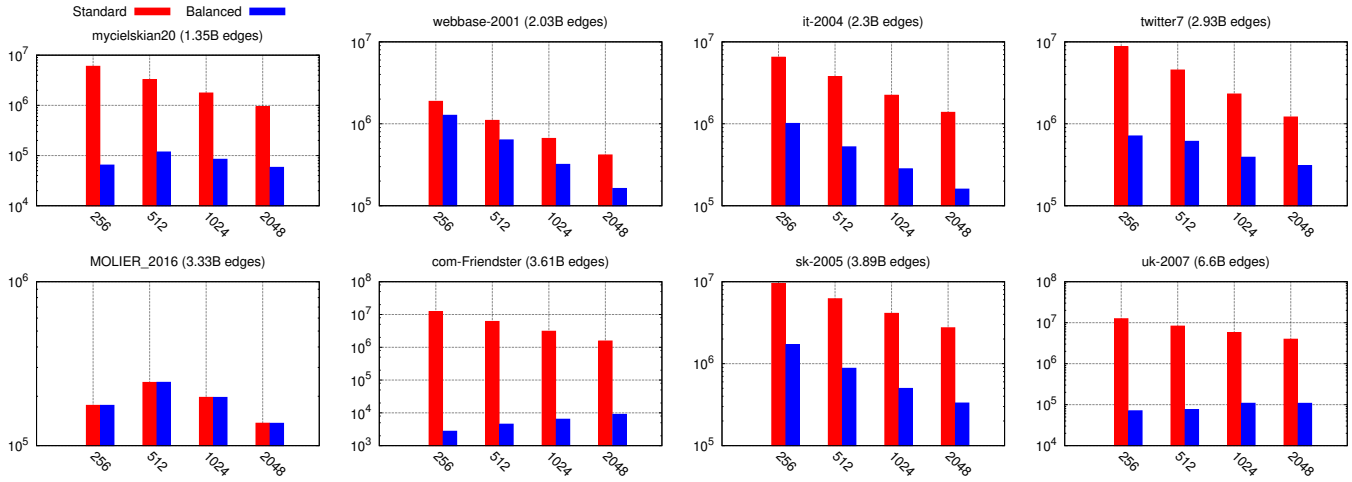

Fig. 2. Graph distribution characteristics of standard and edge-balanced vertex-based distribution. *Y-axis*: Standard Deviation (#Edges/process), *X-axis*: #Processes.

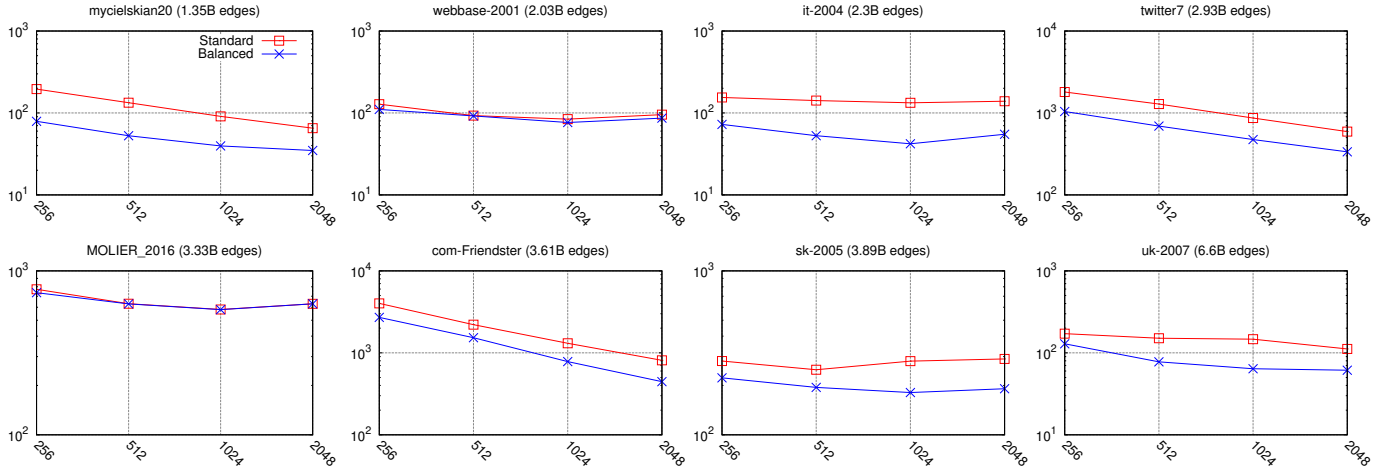

Fig. 3. Distributed-memory Louvain execution times of real-world graphs with over a billion edges, using the standard and edge-balanced graph distribution. *Y-axis*: Total execution time (in secs.), *X-axis*: #Processes.

of iterations to convergence (over 600). It is evident from the %MPI field that for most of the graphs about 90% of time is spent in MPI communication.

TABLE IV
FAST-TRACKING RESISTANCE VS LOUVAIN ON REAL-WORLD GRAPHS WITH A GOOD COMMUNITY STRUCTURE

Name	|V|	|E|	#C(LVN)	#C(FTR)	Q(LVN)	Q(FTR)
soc-flickr	513.9K	6.38M	4362	4391	0.6650	0.6651
ecology2	999.9K	5.99M	172	176	0.9831	0.9835
belgium_osm	1.44M	3.09M	473	499	0.9940	0.9941
CurlCurl_1	226.4K	2.69M	70	79	0.9604	0.9612
denormal	89.4K	1.24M	28	30	0.9229	0.9279
vsp_bump2_e18	56.4K	601.6K	5	7	0.5441	0.5452
sd2010	88.3K	410.7K	47	50	0.9404	0.9423
luxembourg_osm	114.5K	239.3K	227	232	0.9880	0.9883

Finally, we also compare the quality of Fast-Tracking Resistance with Louvain for real-world graphs with a good community structure (and hence, a higher modularity). Table IV shows that even for graphs with a relatively better community structure, Fast-Tracking Resistance can detect more clusters (denoted as #C), and hence enhance the overall modularity (denoted as Q) as compared to Louvain.

ACKNOWLEDGEMENTS

The research is supported in part by the U.S. DOE ExaGraph project, and the NSF award CCF 1815467 to WSU. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75 – 174, 2010.
[2] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "Maximizing modularity is hard," *arXiv preprint physics/0608255*, 2006.

TABLE III
MISCELLANEOUS PERFORMANCE METRICS OF FOUR REAL-WORLD GRAPHS ON 16-128 NODES OF ALCF THETA.

#Nodes	Power(W)	Energy(J)	Mem. per PE (MB)	DDR traffic (GB)	HBM traffic (GB)	Mem. traffic rate (GB/S)	#Edges/S	#Edges/W	%MPI
com-Friendster(3.61B edges)									
16	2.28E+03	6.89E+06	343.2	26,102	45,302	23.61	2.74E+05	3.34E+05	40
32	4.56E+03	8.10E+06	334	11,011	21,897	18.52	5.34E+05	1.79E+05	47.8
64	9.10E+03	1.04E+07	275.7	5,383	11,668	14.84	9.13E+05	9.11E+04	53.6
128	1.83E+04	1.38E+07	317.7	3,453	5,457	11.8	9.54E+05	2.66E+04	64.5
uk-2007 (6.6B edges)									
16	2.21E+03	1.17E+06	127.1	492.35	755.57	2.36	4.06E+06	2.44E+05	92.1
32	4.34E+03	2.13E+06	124	305.99	390.57	1.42	6.26E+06	1.24E+05	95.3
64	8.72E+03	4.03E+06	109.5	268.73	246.94	1.12	7.96E+06	6.17E+04	97.1
128	1.75E+04	8.17E+06	106.4	358.08	167.67	1.13	6.70E+06	2.64E+04	98.2
webbase-2001 (2.03B edges)									
16	2.22E+03	1.19E+06	129.7	434.86	622.7	1.97	1.90E+07	9.87E+05	92.8
32	4.43E+03	2.31E+06	124.2	311.51	373.43	1.31	2.27E+07	4.94E+05	96
64	8.91E+03	4.48E+06	108.4	313.83	220.53	1.06	2.65E+07	2.45E+05	97.7
128	1.78E+04	9.38E+06	104.4	518.71	127.93	1.22	2.21E+07	1.23E+05	98.7
twitter7 (2.93B edges)									
16	2.22E+03	3.80E+06	689.2	7,497	12,912	11.94	1.71E+07	7.97E+06	77.8
32	4.48E+03	5.44E+06	540	3,957	7,927	9.79	2.58E+07	3.96E+06	83.7
64	8.95E+03	7.87E+06	420.1	2,198	4,632	7.77	3.77E+07	1.98E+06	88.1
128	1.80E+04	1.21E+07	303.6	1,356	2,868	6.28	5.28E+07	9.84E+05	91.7

[3] M. E. Newman, "Equivalence between modularity optimization and maximum likelihood methods for community detection," *Physical Review E*, vol. 94, no. 5, p. 052315, 2016.

[4] C. Granell, S. Gomez, and A. Arenas, "Hierarchical multiresolution method to overcome the resolution limit in complex networks," *International Journal of Bifurcation and Chaos*, vol. 22, no. 07, p. 1250171, 2012.

[5] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song *et al.*, "Streaming graph challenge: Stochastic block partition," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–12.

[6] 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 *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 885–895.

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

[8] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

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

[10] 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 *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 885–895.

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

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

[13] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, "memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[14] T. Davis, Y. Hu, and S. Kolodziej, "The suitesparse matrix collection," 2018.

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

978-1-7281-5020-8/19/$31.00 ©2019 IEEE