2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)

`miniVite`: A Graph Analytics Benchmarking Tool for Massively Parallel Systems

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*—Benchmarking of high performance computing systems can help provide critical insights for efficient design of computing systems and software applications. Although a large number of tools for benchmarking exist, there is a lack of representative benchmarks for the class of irregular computations as exemplified by graph analytics. In this paper, we propose `miniVite` as a representative graph analytics benchmark tool to test a variety of distributed-memory systems.**

Graph clustering, popularly known as community detection, is a prototypical graph operation used in numerous scientific computing and analytics applications. The goal of clustering is to partition a graph into clusters (or communities) such that each cluster consists of vertices that are densely connected within the cluster and sparsely connected to the rest of the graph. Modularity optimization is a popular technique for identifying clusters in a graph. Efficient parallelization of modularity optimization-based algorithms is challenging. One successful approach was conceived in `Vite`, a distributed-memory implementation of the Louvain algorithm that incorporates several heuristics.

We introduce `miniVite` as a representative but simplified variant of `Vite`, to serve as a prototypical graph analytics benchmarking tool. Unlike other graph-based methods such as breadth-first search and betweenness centrality, `miniVite` represents highly complex computational patterns stressing a variety of system features, which can provide crucial insight for co-design of future computing systems.

I. Introduction

Evaluating the performance of real-world applications on high performance computing architectures is an important activity [11, 21] that not only leads to efficient implementations, but also allows to improve hardware features in support of the application development. The importance of mini-app driven co-design of architectures and algorithms has been established as a holistic approach in assessing key performance issues in large scientific applications [2, 9, 10, 14]. However, a significant number of mini-apps used in HPC co-design are characterized by regular updates to dense data structures such as meshes and matrices. Hence, there is an urgent need to explore mini-apps characterized by irregular memory accesses, which is the mainstay of a large number of graph applications. Availability of large scale datasets [4, 15, 20] has led to the emergence of graph analytics as an important activity on modern computer systems. However, due to the irregular nature of memory accesses, high ratios of communication to computation and inherently serial nature of execution, graph algorithms pose significant challenges for efficient implementation on parallel systems [16]. We propose `miniVite` as a benchmarking tool for distributed-memory parallel systems.

Consider an undirected graph $G = (V, E, \omega)$, where V is the set of vertices, E the set of edges and ω the edge weights. A clustering C of G is a partitioning of V into k mutually disjoint clusters such that the vertices in a cluster $C_i, \forall i \in k$ are tightly connected with other vertices in C_i but sparsely connected to the rest of the graph. Clustering, popularly known as community detection, is an important graph kernel used in a number of scientific and social networking applications for discovering higher order structures within a graph [12]. In an earlier work, we developed `Vite`[1] [13], which is a distributed memory parallelization of the widely used Louvain method [3] for graph clustering. In this work, we present a variant of `Vite` as a benchmarking tool for distributed-memory systems in the context of graph analytics.

We make the following contributions in this paper: (a) Develop `miniVite` as a prototypical graph analytics benchmarking tool (§II); (b) Characterize the key features of `miniVite` and compare it with other graph workloads (§III); (c) Compare the relative performances of different MPI communication mechanisms (such as collective, send-recv and RMA) used for implementing communication intensive parts of `miniVite` (§V); (d) Analyze the performance of `miniVite` on 1-4K processes of NERSC Cori supercomputer (§V); and (e) Develop an efficient distributed-memory graph generator using the random geometric graph (RGG) model for evaluating `miniVite` (§IV).

II. Parallel Clustering Algorithm

In this section, we provide a brief overview of the parallel implementation of the Louvain method. We refer the reader to Blondel *et al.* [3] for details on the serial algorithm, and Ghosh *et al.* [13] for details on the distributed-memory implementation.

The Louvain algorithm is a heuristic to identify a community-wise partitioning of vertices in a graph that optimizes modularity [18]. The algorithm is multi-phase, multi-iterative, where within each phase there are multiple iterations, as summarized in Algorithm 1. Each vertex is initialized to a distinct community. Within each iteration, each vertex makes a greedy decision on whether to stay in its current community

[1] Source code: http://hpc.pnl.gov/people/hala/grappolo.html.

978-1-7281-0182-8/18/$31.00 ©2018 IEEE
DOI 10.1109/PMBS.2018.00009

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 a "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 algorithm terminates when the net gain in modularity falls below a certain threshold.

Algorithm 1: Parallel Louvain Algorithm (at rank i).
Input: Local portion $G_i = (V_i, E_i)$ of the graph G; 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
  end if
  $NextPhase(G_i, C_{curr})$
  $Q_{prev} \leftarrow Q_{curr}$
end while

Initially, the set of vertices in the graph is evenly partitioned in a trivial manner, among the p processes. Edges are distributed based on the vertex partitioning. Thus, 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. Similarly, the set of communities is also partitioned in the same manner where each processor receives roughly the same number of communities at the start. Note that at the start of the Louvain algorithm the number of communities is same as the number of vertices in the graph. Thus, each process maintains two lists, one for its ghost vertices and another for its ghost communities (along with their owning process IDs).

a) Louvain iteration: Algorithm 2 lists the steps for performing a sequence of Louvain iterations within a phase. Since each process owns a subset of vertices and a subset of communities, communication usually involves information on vertices and/or communities. For each vertex owned locally, a community ID is stored; and for each community owned locally, its incident degree is stored locally (as part of the vector C_{info} in Algorithm 2). After every iteration (within a phase), changes to the community membership information need to be relayed from the corresponding owner processes to all those processes that keep a ghost copy of those communities.

`miniVite` implements the very first phase of the Louvain method, without rebuilding the graph. This allows us to accurately assess the overhead of community detection separately from the graph rebuilding process. `miniVite` can also

Algorithm 2: Algorithm for the Louvain iterations of a phase at rank i. **Output**: Modularity at the end of the phase.

function LOUVAINITERATION(G_i, C_{curr})
$V_g \leftarrow$ Exchange ghost vertices
while true **do**
  Exchange latest community information
  for $v \in V_i$ **do**
    Compute ΔQ achieved by moving v to its neighboring communities
    Choose the community assignement for v that maximizes ΔQ
    Update community information after v's migration
  end for
  send updated information on ghost communities
  $C_{info} \leftarrow$ `receive` and update information on local communities
  $Q^i_{curr} \leftarrow$ Compute modularity based on G_i and C_{info}
  $Q_{curr} \leftarrow$ `all-reduce`: $\sum_{\forall i} Q^i_{curr}$
  if $Q_{curr} - Q_{prev} \leq \tau$ **then**
    break
  end if
  $Q_{prev} \leftarrow Q_{curr}$
end while
return Q_{prev}

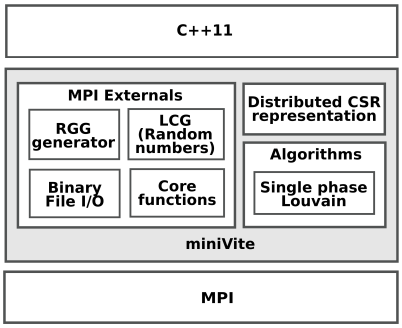


Fig. 1: Components of `miniVite` proxy application.

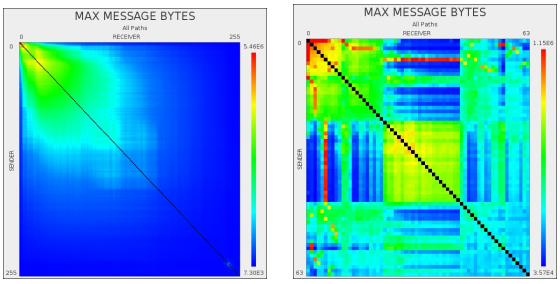
generate random geometric graphs (RGG) in parallel, thereby making it convenient for users to parameterize synthetic graphs (with different communication characteristics) to run the Louvain algorithm. A high level diagram of `miniVite` proxy application is shown in Fig. 1. The common components in `miniVite` such as distributed compressed sparse row (CSR) representation for graphs [8], data generator, and parallel random number generator provide the requisite infrastructure to develop other graph algorithms in the framework.

III. CHARACTERISTICS OF DISTRIBUTED-MEMORY LOUVAIN METHOD

We argue that community detection is a better tool for benchmarking irregular applications, because it exhibits different characteristics in comparison to other graph traversal based workloads. For example, the Louvain method involves floating point arithmetic operations for computing modularity, whereas other graph algorithms such as breadth-first search and graph coloring do not have any floating point operations. `miniVite` is also communication-intensive, as within every Louvain

TABLE I: First phase of Louvain method versus the last phase for real world inputs on 1K processes of NERSC Cori.

Graphs	#Vertices	#Edges	First phase			Complete execution			
			Iterations	Modularity	Time	Phases	Iterations	Modularity	Time
friendster	65.6M	1.8B	143	0.619	565.201	3	440	0.624	567.173
it-2004	41.3M	1.15B	14	0.394	45.064	4	91	0.973	45.849
nlpkkt240	27.9M	401.2M	3	0.143	3.57	5	832	0.939	21.084
sk-2005	50.6M	1.9B	11	0.314	71.562	4	83	0.971	72.94
orkut	3M	117.1M	89	0.643	59.5	3	281	0.658	59.64
sinaweibo	58.6M	261.3M	3	0.198	270.254	4	108	0.482	281.216
twitter-2010	21.2M	265M	3	0.028	209.385	4	184	0.478	386.483
uk2007	105.8M	3.3B	9	0.431	35.174	6	139	0.972	37.988
web-cc12-paylvladmin	42.8M	1.2B	31	0.541	140.493	4	159	0.687	146.92
webbase-2001	118M	1B	14	0.458	14.702	7	239	0.983	24.455



(a) Friendster (1.8B edges) on 256 processes.

(b) Orkut (117M edges) on 64 processes.

Fig. 2: Communication volume, in terms of maximum `send`/`recv` message sizes (bytes) exchanged between pairs of processes, for two real-world inputs. The vertical axis represents the sender process ids and the horizontal axis represents the receiver process ids; the top-left corner represents id zero for both sender and receiver. Byte sizes vary from 7.30E3 (blue) to 5.46E6 (red) for Friendster, and from 3.57E4 (blue) to 1.15E6 (red) for Orkut.

iteration, information of ghost communities (such as current size and degree of communities) needs to be updated for computing global modularity. As such, the overall performance of `miniVite` is sensitive to the input graph (especially since our simple graph partitioning makes no assumption about the underlying graph structure). Fig. 2 shows the inter-process communication volume of `miniVite` for two inputs on 256 and 64 processes respectively, and they exhibit significantly different communication patterns.

Two conflicting goals – simplicity of the benchmark, and true representation of real-world applications – drive the choice of a good benchmarking tool. The following two observations from the performance analysis of `Vite` led us to the design of `miniVite` as a potential benchmarking proxy application for the Exascale Computing Project[2].

a) Louvain phase analysis: Although the Louvain method is executed for multiple phases until convergence, for a variety of real world inputs, we observed the first phase to be the most expensive in terms of overall execution time. Table I demonstrates that most of the input graphs exhibit a cumulative difference of about $1-5\%$ between the execution times of the first and the final phase. Therefore, analyzing just the first phase provides sufficient information about the overall performance and community structure in most cases. Furthermore, graph rebuilding kernel complicate the implementation and can distort benchmarking results when the graph sizes are small and utilize a small portion of the total participating processors on a system.

b) Performance profiling: We profiled `Vite` extensively using HPCToolkit [1] on a billion-edge graph, and observed that about 60% of the time was spent in managing and communicating vertex-community information, and, about 40% was spent on computation/communication (i.e., `MPI_Allreduce`) of global modularity. Profiling helped us in identifying communication intensive sites in the application, where we can apply alternate communication options such as MPI collectives or RMA (Remote Memory Access) and measure their impact.

IV. Synthetic Data Generation

A key component of `miniVite` is the distributed-memory parallel random geometric graph (RGG) generation. The generator allows us to bypass the file I/O for reading an input graph and create a synthetic graph in memory, which can be further parameterized to affect the overall communication intensity. We specifically chose RGGs because they are known to naturally exhibit consistent community structure with high modularity [6], as opposed to scale-free graphs.

An n-D random geometric graph (RGG), represented as $G(n,d)$, is a graph generated by randomly placing N vertices in an n-D space and connecting pairs of vertices whose Euclidean distance is less than or equal to d. In our experiments we only consider 2D RGGs contained within a unit square, $[0,1]^2$, and the Euclidean distance between two vertices is used as the weight of the edge connecting them. We calculate d from two quantities, as explained next. Connectivity is a monotonic property of RGG, in 2D unit-square RGGs have a sharp threshold at $d_c = \sqrt{\frac{\ln N}{\pi N}}$ [7]. The connectivity threshold is also the longest edge length of the minimum spanning tree in G [19]. The thermodynamic limit when a giant component appears with high probability is given by $d_t = \sqrt{\frac{\lambda_c}{\pi N}}$ [7], and the value of λ_c is given by 2.0736 for 2D unit-square RGGs. The particular value of d that we have used in `miniVite` is $d_{ct} = (d_c + d_t)/2$.

We distribute the domain such that each process receives N/p vertices (where p is the total number of processes). Each process owns $1 \times \frac{1}{p}$ of the unit square, and generates that many random numbers, between specific ranges, as shown in Figure 3.

[2]ECP Proxy Applications: https://proxyapps.exascaleproject.org

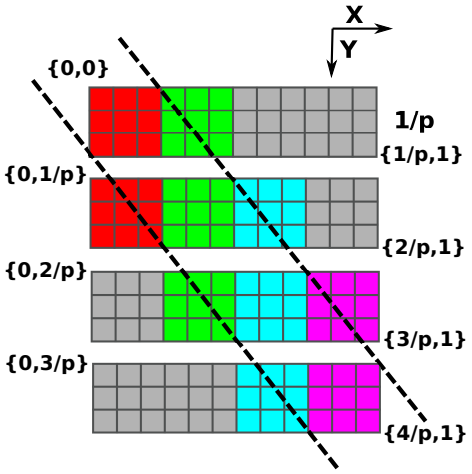
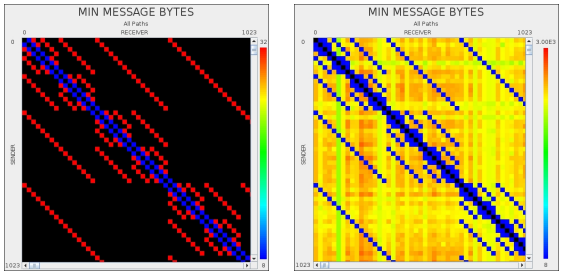


Fig. 3: Distribution based on $[0,1]^2$ on $p = 4$ processes and for $N = 12$. $\frac{1}{p} > d$ mandates that vertices in a process can only have edges with vertices owned by its *up* or *down* neighbor. The blocks between the parallel lines indicate vertices owned by a process.



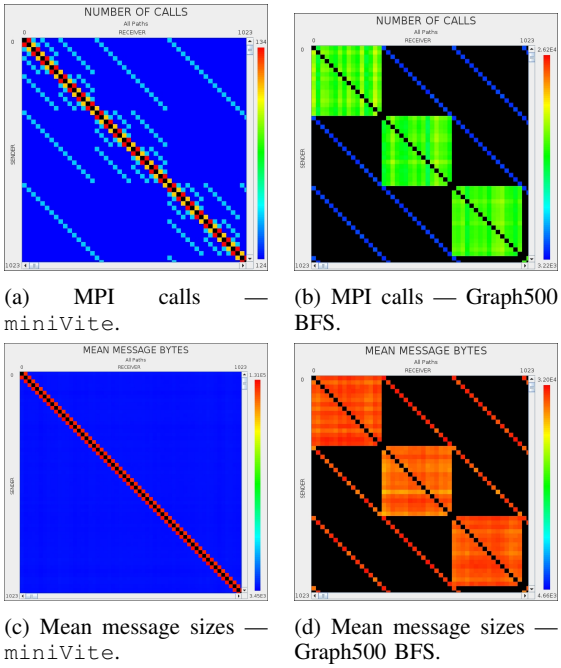
(a) Basic RGG input, black spot means zero exchange.

(b) RGG input with 20% extra edges.

Fig. 4: Communication volume, in terms of minimum `send/recv` message sizes (in bytes) exchanged between pairs of processes, of `miniVite` with basic RGG input vs RGG with random edges using 1024 processes. Adding extra edges increase overall communication. The vertical axis represents the sender process ids and the horizontal axis represents the receiver process ids; the top-left corner represents id zero for both sender and receiver. Byte sizes vary from 8 (blue) to 32 (red) for the figure on left, and from 8 (blue) to 3000 (red) for the figure on right.

The generated random numbers are exchanged between neighboring processes, in order to compute Euclidean distance between neighboring vertices. The ghost vertices are then exchanged between neighbors. Since RGG relies on random numbers, it is important that the sequence of numbers be chosen from the same distribution across processes. We implement the linear congruential generator (LCG) algorithm in distributed memory. LCG is defined by a linear recurrence relation to deterministically generate a sequence of random numbers.

`miniVite` also provides an option for introducing some noise into a RGG, by adding a percentage of total edges randomly (following a uniform distribution) between vertices. Adding random edges increases the likelihood of a process communicating with other non-neighboring processes, increasing the overall network congestion, and thereby creating ideal scenarios for measuring the impact of different communication options. Figure 4 shows inter-process communication (as reported by TAU profiler [22]) in `miniVite` between 1024 processes of NERSC Cori, using basic RGG as compared to RGG with 20% random edges for a graph of 134M vertices and 1.6B edges.



(a) MPI calls — `miniVite`.

(b) MPI calls — Graph500 BFS.

(c) Mean message sizes — `miniVite`.

(d) Mean message sizes — Graph500 BFS.

Fig. 5: Communication volumes (in terms of `send/recv` invocations, and mean `send/recv` message sizes exchanged between processes) of `miniVite` and Graph500 BFS for 134M vertices on 1024 processes. Black spots indicate zero communication. The vertical axis represents the sender process ids and the horizontal axis represents the receiver process ids; the top-left corner represents id zero for both sender and receiver. Blue represents the minimum and Red represents maximum volume for each of the figures at different minimum and maximum values (communication patterns are important).

Figure 5 shows inter-process communication patterns between `miniVite` with RGG input of 134M vertices (20% of the overall number of edges added randomly) compared to Graph500 BFS [17] (with `SCALE` equal to 27). It is evident from the figure that only a subset of processes participate in communication for Graph500 BFS, whereas all of the processes contribute to the overall communication in `miniVite`.

V. Preliminary Evaluation

We now present results from a preliminary evaluation of `miniVite`. We use the following notations for different variants of communication: i) **`NBSR`**: Uses MPI nonblocking point-to-point communication routines.ii) **`SR`**: Uses blocking MPI send and receive, i.e., `MPI_Sendrecv`. iii) **`COLL`**: Uses MPI blocking collective operation, uses `MPI_Alltoallv`. iv) **`RMA`**: Uses MPI-3 RMA for one-sided communication with passive target synchronization; uses `MPI_Put` or `MPI_Accumulate` (only for Friendster).

a) Experimental setup: We used the NERSC Cori supercomputer for our experimental evaluations. Each node of Cori has dual-socket Intel® Xeon™E5- 2698v3 (Haswell) CPUs at 2.3 GHz, 32 cores, 128 GB main memory, 40 MB L3

TABLE II: Execution time (in secs.) and Modularity (Q) on 1-4K processes for RGG datasets with unit edge weights.

Versions	1024 processes ($\|V\|$ = 134.2M)				2048 processes ($\|V\|$ = 268.4M)				4096 processes ($\|V\|$ = 536.8M)			
	$\|E\|$ = 1.59B		$\|E\|$ = 1.9B		$\|E\|$ = 3.24B		$\|E\|$ = 3.89B		$\|E\|$ = 6.64B		$\|E\|$ = 7.97B	
	Time	Q	Time	Q	Time	Q	Time	Q	Time	Q	Time	Q
NBSR	6.53	0.750	18.28	0.626	9.57	0.749	21.68	0.626	49.53	0.748	57.06	0.625
COLL	5.56	0.750	18.32	0.626	7.28	0.749	21.65	0.626	18.10	0.748	47.85	0.625
SR	13.30	0.750	28.32	0.626	31.50	0.749	49.27	0.626	94.41	0.748	115.87	0.625
RMA	5.76	0.751	19.05	0.626	8.82	0.753	23.23	0.626	47.18	0.750	60.95	0.626

TABLE III: Execution time (in secs.) and Modularity (Q) on 1-4K processes for RGG datasets with Euclidean distance weights.

Versions	1024 processes ($\|V\|$ = 134.2M)				2048 processes ($\|V\|$ = 268.4M)				4096 processes ($\|V\|$ = 536.8M)			
	$\|E\|$ = 1.59B		$\|E\|$ = 1.9B		$\|E\|$ = 3.24B		$\|E\|$ = 3.9B		$\|E\|$ = 6.64B		$\|E\|$ = 7.97B	
	Time	Q	Time	Q	Time	Q	Time	Q	Time	Q	Time	Q
NBSR	5.99	0.776	13.08	0.648	9.54	0.776	17.50	0.629	30.46	0.776	20.15	0.599
COLL	5.46	0.776	12.87	0.653	7.46	0.776	14.17	0.628	15.34	0.776	21.83	0.598
SR	13.17	0.776	19.6	0.649	32.05	0.776	31.97	0.624	88.81	0.776	65.17	0.598
RMA	5.80	0.777	13.3	0.628	9.18	0.776	15.43	0.624	21.96	0.776	22.41	0.544

cache/socket and the Cray® XC™series interconnect (Cray® Aries™with Dragonfly topology). We use Cray MPICH 7.6.2 as our MPI implementation, and Intel® 18.0.1 compiler with `-O3 -xHost` compilation option to build the codes. We use 16 processes per node, and 2 OpenMP [5] threads per process. We only report the execution time for graph clustering using the Louvain method.

A. Performance on a real-world graph—Friendster

For our analysis, we chose a real-world network that runs for a number of iterations, and has a reasonable community structure. Friendster is a social network graph, with 1.8B edges and a modularity of about 0.62. In our studies, it ran for over 400 iterations until convergence, with each phase taking more than a 100 iterations (exhibiting relatively slow modularity growth), making Friendster an ideal real-world dataset for clustering analysis.

TABLE IV: Number of iterations, execution time (in secs.) and Modularity (Q) of Friendster (65.6M vertices, 1.8B edges) on 1024/2048 processes.

Versions	1024 processes			2048 processes		
	Itrs	Time	Q	Itrs	Time	Q
NBSR	111	745.80	0.6155	127	498.89	0.6177
COLL	109	752.41	0.6159	141	554.98	0.6204
SR	111	783.94	0.6157	103	423.43	0.6191
RMA	109	782.47	0.6162	111	589.47	0.6190

From Table IV, we observe both `NBSR`/`SR` to outperform `COLL`/`RMA` by at least $10-20\%$ for 2K processes. For 1K processes, the performances of `NBSR` and `COLL` are competitive, whereas `SR` and `RMA` are about 5% slower. The primary reason behind these fluctuations in runtime performances are due to dissimilar number of iterations to convergence ($103-141$), as shown in Table IV, which is a side effect of distributed memory implementation of Louvain method. The `RMA` version with `MPI_Put` failed with a crash (observed on all available Cray MPICH versions on NERSC Cori) for the Friendster input. Hence, `MPI_Put` was replaced with `MPI_Accumulate`, which has different ordering semantics, increasing the overall execution time. Since the Louvain method is inherently sequential, the order of community updates from processes impacts the overall number of iterations to convergence, making it nondeterministic across program runs. This behavior makes it difficult to accurately measure the effect of using different communication models on real-world graphs.

B. Performance on random geometric graphs

Due to the distribution of a random geometric graph (RGG) (refer to §IV), if random edges are not added, then a process communicates with at most two neighboring processes (see Figure 4). Therefore, we also discuss performance of the RGG datasets with additional random edges (20% of the total number of edges, about $0.4-1.3$B).

Table II shows the performance of `miniVite` on RGG datasets of over a billion edges with unit edge weights on 1K-4K processes. When extra edges are added, execution times increase by up to $3\times$, owing to an increase in overall communication volume. However, the change in modularity is more gradual across multi-process runs, and it declines by about 17% when extra edges were added.

We also analyzed the impact of using real edge weights (the Euclidean distance) between vertices, as shown in Table III. If the Euclidean distance between a randomly selected vertex pair is unavailable (when the respective vertices are owned by non-neighboring processes), then we pick an edge weight uniformly between $(0, 1)$. We ensure the edge weight is consistent across MPI communication models by providing a seed to the random number generator, equivalent to a unique hash of the vertex pair. Unlike the extra edge cases in Table II, in Table III we observe about $3-9\%$ variability in modularity across MPI communication models.

In general, for the RGG graphs, comparing different MPI communication models, performance of `SR` is at least $1.5-2\times$ worse than others for every case, potentially due to internal message ordering overheads. Performance of `COLL` is consistently superior, and for the largest RGG of 536.8M vertices, `COLL` is about $1.2-5\times$ faster than the rest. Performance of basic RGG datasets (with no extra edges) with unit and real edge weights are comparable, overall modularity difference being about $3-6\%$. However, we observe a significant difference between the two approaches for the largest case (536.8M vertices) with the extra edges (a reduction of about $1.2-3\times$ in execution time and up to 22% in modularity). Due to a significant reduction in number of iterations to convergence, the execution time is lesser than the basic case.

ACKNOWLEDGMENT

We used resources of the NERSC facility, supported by U.S. DOE SC under Contract No. DE-AC02-05CH11231. The research is in part supported by the U.S. DOE ExaGraph project, a collaborative effort of U.S. DOE SC and NNSA at DOE PNNL, NSF CAREER award IIS-1553528, NSF award CCF 1815467, and, ECP (17-SC-20-SC). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

We thank David Richards of Lawrence Livermore National Laboratory for motivating us to develop `miniVite`.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Richard F Barrett, Paul S Crozier, DW Doerfler, Michael A Heroux, Paul T Lin, HK Thornquist, TG Trucano, and Courtenay T Vaughan. Assessing the role of mini-applications in predicting key performance characteristics of scientific and engineering applications. *Journal of Parallel and Distributed Computing*, 75:107–122, 2015.

[3] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, page P10008., 2008.

[4] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[5] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[6] Erik Davis and Sunder Sethuraman. Consistency of modularity clustering on random geometric graphs. *arXiv preprint arXiv:1604.03993*, 2016.

[7] Josep Díaz, Dieter Mitsche, and Xavier Pérez-Giménez. Large connectivity for dynamic random geometric graphs. *IEEE Transactions on Mobile Computing*, 8(6):821–835, 2009.

[8] Jack Dongarra. Compressed row storage. http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html.

[9] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[10] Sudip S Dosanjh, Richard F Barrett, DW Doerfler, Simon D Hammond, Karl S Hemmert, Michael A Heroux, Paul T Lin, Kevin T Pedretti, Arun F Rodrigues, TG Trucano, et al. Exascale design space exploration and co-design. *Future Generation Computer Systems*, 30:46–58, 2014.

[11] Rudolf Eigenmann and Siamak Hassanzadeh. Benchmarking with real industrial applications: the spec high-performance group. *IEEE Computational Science and Engineering*, 3(1):18–23, 1996.

[12] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75 – 174, 2010.

[13] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895. IEEE, 2018.

[14] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[16] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17:5–20, 2007.

[17] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[18] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[19] Mathew Penrose et al. *Random geometric graphs*. Number 5. Oxford university press, 2003.

[20] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, volume 15, pages 4292–4293, 2015.

[21] M. Sayeed, H. Bae, Y. Zheng, B. Armstrong, R. Eigenmann, and F. Saied. Measuring high-performance computing with real applications. *Computing in Science Engineering*, 10(4):60–70, July 2008.

[22] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.