**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}@pnl.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 benchmarks for the class of irregular computations 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 $\omega$ 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 [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 (§IV); (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

---

1Source code: http://hpc.pnl.gov/people/hala/grappolo.html.
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 \( \tau \)). 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 \( \tau \) (default: \( 10^{-6} \)).

**Notation:** \( C \) denotes communities, and \( Q \) denotes modularity.

\[
1: C_{\text{curr}} \leftarrow \{\{u|\forall u \in V\}\}
2: \{Q_{\text{curr}}, Q_{\text{prev}}\} \leftarrow 0
3: \text{while true do}
4: \quad Q_{\text{curr}} \leftarrow \text{LouvainIteration}(G_i, C_{\text{curr}})
5: \quad \text{if } Q_{\text{curr}} - Q_{\text{prev}} \leq \tau \text{ then}
6: \quad \quad \text{break and output the final set of communities}
7: \quad \text{end if}
8: \quad \text{NextPhase}(G_i, C_{\text{curr}})
9: \quad Q_{\text{prev}} \leftarrow Q_{\text{curr}}
10: \text{end while}
\]

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 \( C_{\text{info}} \) 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.

\texttt{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. \texttt{miniVite} can also 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 \texttt{miniVite} proxy application is shown in Fig. 1. The common components in \texttt{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. \texttt{miniVite} is also communication-intensive, as within every Louvain

---

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

\[
1: \quad \text{function LOUVAINITERATION}(G_i, C_{\text{curr}})\]
\[
2: \quad V_g \leftarrow \text{Exchange ghost vertices}
3: \quad \text{while true do}
4: \quad \quad \text{Exchange latest community information}
5: \quad \quad \text{for } \forall v \in V_i \text{ do}
6: \quad \quad \quad \text{Compute } \Delta Q \text{ achieved by moving } v \text{ to its neighboring communities}
7: \quad \quad \quad \text{Choose the community assignment for } v \text{ that maximizes } \Delta Q
8: \quad \quad \quad \text{Update community information after } v \text{'s migration}
9: \quad \text{end for}
10: \quad \text{send updated information on ghost communities}
11: \quad C_{\text{info}} \leftarrow \text{receive and update information on local communities}
12: \quad Q'_{\text{curr}} \leftarrow \text{Compute modularity based on } G_i \text{ and } C_{\text{info}}
13: \quad Q_{\text{curr}} \leftarrow \text{all-reduce: } \sum_i Q'_i
14: \quad \text{if } Q_{\text{curr}} - Q_{\text{prev}} \leq \tau \text{ then}
15: \quad \quad \text{break}
16: \quad \quad \text{end if}
17: \quad Q_{\text{prev}} \leftarrow Q_{\text{curr}}
18: \text{end while}
19: \quad \text{return } Q_{\text{prev}}
\]
TABLE I: First phase of Louvain method versus the last phase for real world inputs on 1K processes of NERSC Cori.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>First phase</th>
<th>Complete execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Iterations</td>
<td>Modularity</td>
</tr>
<tr>
<td>friendster</td>
<td>65.6M</td>
<td>1.3B</td>
<td>143</td>
<td>0.619</td>
</tr>
<tr>
<td>it-2004</td>
<td>41.3M</td>
<td>1.13B</td>
<td>14</td>
<td>0.394</td>
</tr>
<tr>
<td>npkklz40</td>
<td>27.9M</td>
<td>401.2M</td>
<td>3</td>
<td>0.143</td>
</tr>
<tr>
<td>sk-2005</td>
<td>50.6M</td>
<td>1.9B</td>
<td>11</td>
<td>0.314</td>
</tr>
<tr>
<td>orkut</td>
<td>3M</td>
<td>117.1M</td>
<td>89</td>
<td>0.643</td>
</tr>
<tr>
<td>sinaweibo</td>
<td>58.6M</td>
<td>261.3M</td>
<td>3</td>
<td>0.198</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>21.2M</td>
<td>265M</td>
<td>3</td>
<td>0.028</td>
</tr>
<tr>
<td>nk2007</td>
<td>105.8M</td>
<td>3.3B</td>
<td>9</td>
<td>0.431</td>
</tr>
<tr>
<td>web-cc12-payvladmin</td>
<td>42.8M</td>
<td>1.2B</td>
<td>31</td>
<td>0.541</td>
</tr>
<tr>
<td>webbase-2001</td>
<td>118M</td>
<td>1B</td>
<td>14</td>
<td>0.458</td>
</tr>
</tbody>
</table>

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.

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 $d_c = \sqrt{\ln N / \pi N}$ [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 $d_t = \sqrt{\lambda_c / \pi N}$ [7], and the value of $\lambda_c$ 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 $1 \times \frac{d}{2}$ of the unit square, and generates that many random numbers, between specific ranges, as shown in Figure 3.
$\frac{1}{p} > d$ 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.

![Fig. 3: Distribution based on $[0, 1]^2$ on $p = 4$ processes and for $N = 12$.](image)

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.

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

![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 3200 (red) for the figure on left, and from 8 (blue) to 3000 (red) for the figure on right.](image)

![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. 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.](image)
cache/socket and the Cray® XCTM series interconnect (Cray® AriesTM 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 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.

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 – 1.3B).

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×, 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× 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× 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× 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.

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.
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