Parallel Computing 111 (2022) 102898

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Towards scaling community detection on distributed-memory heterogeneous systems

Nitin Gawande [a,1], Sayan Ghosh [a,*], Mahantesh Halappanavar [a], Antonino Tumeo [a], Ananth Kalyanaraman [b]

[a] Pacific Northwest National Laboratory, Richland, WA, United States of America
[b] Washington State University, Pullman, WA, United States of America

ARTICLE INFO

Keywords:
Distributed community detection
Heterogeneous systems
Multi-GPU
Parallel Louvain
Parallel graph algorithms

ABSTRACT

In most real-world networks, nodes/vertices tend to be organized into tightly-knit modules known as *communities* or *clusters* such that nodes within a community are more likely to be connected or related to one another than they are to the rest of the network. Community detection in a network (graph) is aimed at finding a partitioning of the vertices into communities. The goodness of the partitioning is commonly measured using *modularity*. Maximizing modularity is an NP-complete problem. In 2008, Blondel et al. introduced a multi-phase, multi-iteration heuristic for modularity maximization called the *Louvain* method. Owing to its speed and ability to yield high quality communities, the Louvain method continues to be one of the most widely used tools for serial community detection.

Distributed multi-GPU systems pose significant challenges and opportunities for efficient execution of parallel applications. Graph algorithms, in particular, have been known to be harder to parallelize on such platforms, due to irregular memory accesses, low computation to communication ratios, and load balancing problems that are especially hard to address on multi-GPU systems.

In this paper, we present our ongoing work on distributed-memory implementation of Louvain method on heterogeneous systems. We build on our prior work parallelizing the Louvain method for community detection on traditional CPU-only distributed systems without GPUs. Corroborated by an extensive set of experiments on multi-GPU systems, we demonstrate competitive performance to existing distributed-memory CPU-based implementation, up to 3.2× speedup using 16 nodes of OLCF Summit relative to two nodes, and up to 19× speedup relative to the NVIDIA RAPIDS® cuGraph® implementation on a single NVIDIA V100 GPU from DGX-2 platform, while achieving high quality solutions comparable to the original Louvain method. To the best of our knowledge, this work represents the first effort for community detection on distributed multi-GPU systems. Our approach and related findings can be extended to numerous other iterative graph algorithms on multi-GPU systems.

1. Introduction

Consider a graph $G = (V, E, \omega)$, where V represents a set of vertices or entities, E represents a set of edges or binary relationships on V, and ω represents positive weights associated with edges. Graph clustering or community detection is the problem of partitioning the vertex set V into subsets called communities or clusters such that vertices within a community are tightly connected with each other, while vertices across communities are sparsely connected with each other. There are variants of the problem such as overlapping clustering that allow vertices to be part of multiple clusters, and hierarchical clustering that builds a hierarchical tree of association of how groups or pairs of vertices are clustered [1]. Modularity [2] is a common metric to measure the goodness or quality of clustering. Algorithms based on modularity optimization are effective but proved to be NP-hard [3]. There are also a diverse set of algorithms to solve the problem [4]. In this paper, we focus on one particular algorithm based on the idea of modularity optimization that will be discussed in Section 3.

Graph theoretic modeling is used in numerous applications to comprehensively capture complex interactions between entities such as interacting atoms in a molecule or proteins in an organism. With the

* Corresponding author.
E-mail addresses: nitin.gawande@intel.gov (N. Gawande), sg0@pnnl.gov (S. Ghosh), hala@pnnl.gov (M. Halappanavar), antonino.tumeo@pnnl.gov (A. Tumeo), ananth@wsu.edu (A. Kalyanaraman).
[1] Intel Corporation, Santa Clara CA; work done while at PNNL.

https://doi.org/10.1016/j.parco.2022.102898
Received 4 November 2020; Received in revised form 16 September 2021; Accepted 28 January 2022
Available online 22 February 2022

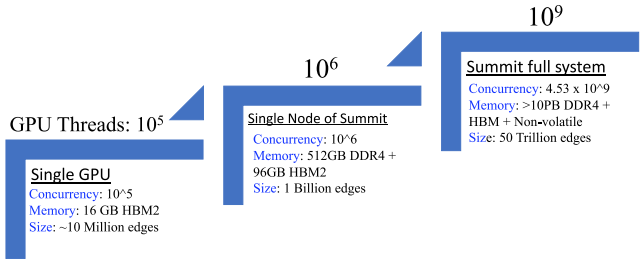


Fig. 1. The scale of GPU thread concurrency and system memory at various levels of Summit hosted at the Oak Ridge Leadership Computing Facility.

ability to discover structurally coherent modules in a graph, community detection has emerged as a fundamental tool in a number of scientific and industrial applications, including biological sciences, computational chemistry, climate sciences, graph analytics, social networks, cyber security, financial networks, and literature mining. Consequently, the need to perform community detection on large scale inputs has become critically important. Comprehensive reviews on the various formulations, methods, and applications of community detection can be found in [4–7]. Clustering is closely related to the problem of graph partitioning, a common and well defined problem in scientific computing with applications such as balanced work distribution among parallel processors and optimization of circuit layouts [8]. Given a graph G and p partitions as input, the objective of graph partitioning is to partition the vertices in G into p partitions such that each partition has roughly the same number of vertices and the edges between any two partitions (called edgecut) is minimized. Thus, the two key distinctions between partitioning and clustering are that the number of clusters is not known a priori, and that the identified clusters can have different sizes. Consequently, the methods for clustering and partitioning can be fundamentally different.

On the other hand, the push to breach the exascale barrier in computing influenced the emergence of massive scale parallelism through accelerated and heterogeneous computing platforms, in particular, graphic processing units (GPUs) programmable with general purpose code (GPGPU). We provide a brief discussion in Section 2.1. The unprecedented amount of concurrency to the order of billions of hardware threads, and the deep, complex memory hierarchy brought by heterogeneous architectures have emerged as formidable challenges for the design and development of scalable algorithms. In particular, graph algorithms pose special challenges such as inherently sequential algorithms, irregular memory access patterns, low computation to communication ratios, and load imbalances at multiple levels [9,10]. As summarized in Fig. 1, the Summit system at the Oak Ridge Leadership Computing Facility (OLCF) is a prototypical GPU-accelerated petascale system that offers an unprecedented amount of parallelism and system memory.

Using community detection as a prototypical case study, we discuss design choices, and present early results on designing and scaling our algorithm at three levels of hierarchy: single GPU, single node multiple GPUs and multiple nodes multiple GPUs. We discuss some of the practical issues in developing customizable graph applications on CPU/GPU systems arising from disparities in data representations, thread divergence, coalesced memory accesses and load balancing.

We provide a brief overview of the relevant hardware platforms in Section 2.1.

Contributions: We make the following contributions in this paper:

(i) Present cuVite, a distributed multi-GPU C++ library for community detection using the Louvain method as a serial template and detailed in Section 3.
(ii) Discuss the various challenges in porting irregular applications on multi-GPU systems and present strategies to address these challenges; detailed in Sections 3 and 4.
(iii) Demonstrate up to 20× improvement relative to NVIDIA RAPIDS® cuGraph on a single node, and about 1.6–3.2× strong scaling performance over 2–16 Summit nodes (Section 5).
(iv) Demonstrate speedups of up to 6× on 2048 processes of ALCF Theta using eight real-world graphs, including characterization of different memory modes (Section 5).
(v) Demonstrate parity of solutions computed by cuVite with the solutions reported by serial and CPU-only implementations (Section 5.5).

To the best of our knowledge, this work presents the first *distributed-memory multi-GPU* graph community detection using the Louvain method as the serial template. We build on the CPU-only distributed graph community detection in Vite [11,12], and make significant additions to exploit multi-GPU nodes. We believe that the details discussed in this paper will benefit not only researchers poring graph algorithms on forthcoming exascale systems, but also application developers in diverse science domains that employ community detection for discovery and analysis.

2. Preliminaries

In this section, we provide some information on contemporary hardware architectures and introduce the graph community detection problem.

2.1. Hardware overview

HPC systems have 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 chip area and energy. However, network and memory bandwidth are not increasing at the same rate, resulting in unbalanced systems, especially for the memory-bound graph analytics workloads. 3D-stacked memory stacks multiple DRAM dies one on top of the other and interconnects them to a memory controller die at as the level of the stack employing through silicon vias (TSVs), thus providing high bandwidth with relatively low energy costs. HBM (High Bandwidth Memory) is by far the predominant type of 3D stacked Dynamic Random Access Memory (DRAM), a number of contemporary CPUs (e.g., Fujitsu A64FX) and GPUs (e.g., NVIDIA Pascal and Volta, AMD Radeon Vega) platforms 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). Our experiments indicate that HBM-based systems hold promise for improving the performance of graph workloads, particularly in the context of community detection, thanks to the much higher bandwidth provided with respect to conventional double data rate (DDR) DRAM, but still require accurate data structure design due to lower utilization of such bandwidth with fine-grained memory transactions.

In this work we consider three different heterogeneous HPC systems. We consider Argonne Leadership Computing Facility (ALCF) Theta, as a system integrating homogeneous cores with specialized units and precursor of the next generation Aurora exascale supercomputer, the NVIDIA DGX2 system as an example of an heterogeneous node with a very high number of GPUs, and Oak Ridge Leadership Computing Facility (OLCF) Summit as the premier example of a large scale supercomputing system with a moderate to high number of GPUs per node and pre-cursor to the Frontier exascale system.

With respect to the other systems considered in this work, ALCF Theta is based on a manycore processor design (Intel Xeon Phi® Knights Landing® - KNL) that employs simple general purpose multithreaded cores with tightly integrated vector units, but starts exposing key aspects that developers need to take into consideration for the effective

exploitation of a more complicated memory hierarchy that integrates 3D-Stacked memory. A KNL node in Theta consists of 64 multithreaded (4 threads each) relatively simple cores with 512-bit vector units, organized into 32 tiles (2 cores/tile, sharing an L2 cache of 1 MB) in a 2-D layout, a high bandwidth in-package multi-channel DRAM memory of size 16 GB (MCDRAM), and 192 GB of DDR4 main memory. The tiles are connected by a mesh interconnect, and the mesh supports different levels of memory address affinity, known as *clustering* modes.

Both the DGX2 and a Summit node feature multiple NVIDIA Tesla® V100® boards, based on the Volta architecture (GV100 GPU), and exploit the second generation NVLINK® interconnect to its fullest extent. However, they have key differences in the organization of the resources in a node: different host CPUs, different number of GPUs, and different topologies to interconnect GPUs to host CPUs and GPUs together.

A key innovation of the Volta Streaming Multiprocessors (SMs) with respect to previous architectures is the way warps are executed. While instructions for threads are still issued in warps (i.e., for a group of 32 threads), their execution is now independently controlled, speeding up those cases where they diverge. This feature can be readily exploited through the use of cooperative groups, a new way to synchronizes different threads. The GV100 hosts 6 MB of L2 cache and 8 memory controllers at 512-bit width (4096-bit in total) to interface with 4 HBM2 stacks. The GPU runs at 1.3 GHz but supports (boost) clocks up to 1.53 GHz. NVLINK2 provides six links with an aggregate 300GB/s bidirectional bandwidth. The DGX-2 node has two 24-core 2.7 GHz Intel Xeon® Platinum 8168® CPUs, with 32KB/1MB per-core L1/L2 caches and 33MB shared L3 cache, and 1.5TB DDR4 memory, and hosts 16 T V100 GPUs with 32 GB of HBM2 each. Each Summit node, instead, includes two IBM Power9® processors with 22 cores, integrating separate 32KB L1 data and instruction caches and connecting to 512 GB of DDR4 memory. Pairs of cores share a 512KB L2 cache and a 10MB L3 cache. A Summit node hosts a total of 6 T V100 GPUs with 16 GB of HBM2 each. The DGX-2 uses a fully interconnected topology across the 0–16 GPUs with 12 NVSwitches, but they communicate with the Intel processors only through PCI Express. The 6 GPUs per node in Summit, instead, are divided in two NVLINK2 fully interconnected blocks of 3 GPUs and a Power9 processor. The two blocks internally communicate only through the processor interconnect (XBus). Additionally, both the DGX-2 and Summit nodes feature Infiniband EDR network interfaces (8 for the DGX-2, and 2 for Summit). However, we only consider a single node for the DGX-2, hence we do not employ the network interconnect. For Summit, instead, we look at scaling when increasing number of nodes.

Unified Virtual Addressing (UVA) is a software and hardware-supported feature in contemporary NVIDIA architectures that, by enabling peer-to-peer access, allows writing code that directly uses pointers to data allocated on one GPU from another. Newer versions of NVLINK enhance UVA features, introducing support for atomic memory operations and significantly increasing bandwidth between devices, thus making migration of virtual memory pages more practical. Our current implementation does not currently leverage UVA features, because our implementation is heterogeneous, where CPUs perform part of the computation and prepare/transform data structures to facilitate GPU computations. UVA is not directly applicable across the nodes of a (distributed memory) cluster (such as Summit) without using network-enabled RDMAs devices, which can complicate code development. However, on single node dense-GPU platforms like the DGX-2, our approach can readily use libraries like NVIDIA NCCL [13] that allows for mapping message passing primitives to data transfer through NVLINK.

2.2. Graph community detection or clustering

Given a graph $G = (V, E, \omega)$, the objective for community detection is to partition the vertex set $V = \{C_1 \cup C_2 \cup \dots C_k\}$ such that vertices within a community are "tightly" connected with each other and sparsely connected with the rest of the graph. Various measures have been proposed in literature to evaluate the goodness of partitioning produced by an [14–16], and one of the measures is *modularity* that has been used widely [2]. Given a partitioning P of V, modularity denoted by Q, can be intuitively expressed as the difference between the fraction of intra-cluster edges imposed by the partitioning P and the expected fraction in an equivalent but randomly reconnected graph with the same number of vertices, edges and vertex degree distribution. Formally, modularity can be computed as follows:

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

where $m = \sum_{e \in E} \omega(e)$ denotes the sum of the weights of all the edges in the graph, $e_{i \to C}$ denotes the sum of the edge weights for the edges connecting vertex i to vertices in community C, and a_C denotes the sum of the degrees of all the vertices in community C. Modularity as a metric has been studied extensively and has been shown to have limitations as the *resolution limit* problem [17].

From an algorithmic perspective, modularity optimization is an NP-complete problem [18], and therefore practitioners depend on efficient heuristics for maximizing modularity for clustering a given graph. Albeit its limitations, the measure continues to be widely used in practice and has been demonstrated to be competitive with other methods for community detection [4,19]. Algorithms addressing the resolution-limit problem have also been proposed in literature [20]. Many efficient community detection heuristics based on maximizing modularity have been developed over the years, making the analysis of large-scale networks feasible in practice. One such efficient heuristic is the *Louvain* method proposed by Blondel et al. [21]. The method is a multi-phase, multi-iteration heuristic that starts from an initial state of $|V|$ communities (each vertex assigned to a unique community), and iteratively improves the quality of community assignment until the gain in modularity is less than a user-defined threshold value. At this point, all the vertices in a given community are coarsened into a single vertex for consideration in the next phase of execution, and edges are added to represent inter-cluster connectivity in the current phase. The algorithm then iterates until the coarsened graph reaches a given minimum size. From a computation perspective, the operations translate into performing multiple sweeps of the graph (one per iteration) and graph coarsenings between successive phases.

Because of its speed and relatively high quality of output in practice [22], the Louvain method has been widely adopted by practitioners in numerous domains. The Louvain method is inherently sequential, and several efforts have been made in parallelizing the method, as detailed in Section 6. Fig. 2 attempts to characterize Louvain method in the space of contemporary modularity-based graph algorithms.

A subset of the authors of this paper have been involved in several such efforts including multi-threaded [26], single GPU [27], and distributed-memory [11] implementations. In this particular work, we

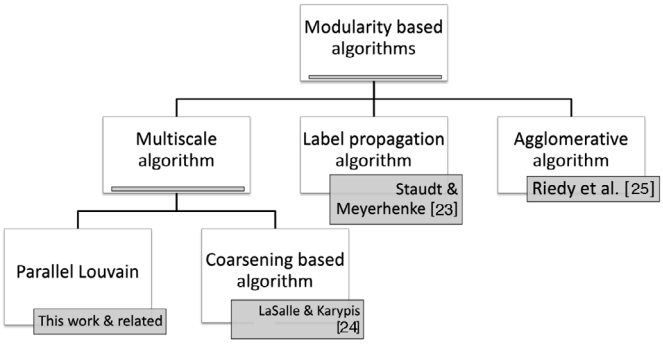


Fig. 2. Louvain Method in the context of modularity based graph algorithms. See Ref. [23–25].

build on the distributed-memory implementation made available as a software package called Vite [11]. The need to port to massively parallel systems emerges from the need to solve large-scale problems, as well as situations when the algorithm needs to be executed a large number of times. For example, Weir et al. use output from multiple executions to compute a better quality solution [28]. Recent work has also demonstrated the application of community detection for ordering of vertices to enhance memory locality [29], and therefore, the speed and accuracy of the algorithm becomes critical.

3. Distributed multi-GPU Louvain method

We provide an overview of cuVite, our distributed multi-GPU community detection algorithm using the Louvain method as a serial template in this section. We provide specific approaches and heuristics for GPU porting in Section 4. We begin with the input graph distribution and provide an overview of the distributed algorithm.

3.1. Input distribution

Due to practical limitations of graph partitioning algorithms, we rely on simple block distribution schemes for partitioning an input graph among participating processes. In particular, we distribute contiguous blocks of vertices and the incident edges across available processes such that each process receives roughly the same number of edges. Each process stores the subset of vertices that it owns. Each process also keeps track of a "ghost" copy for any vertex that has an edge to any of its local vertices but is owned by a different (remote) process. Henceforth, we refer to the latter set of vertices as "ghost" vertices. We use the compressed sparse row (CSR) format to store the vertex and edge lists [30].

Similarly, each process also owns a subset of communities (set initially to an equal number of communities per process), and keeps track of a set of "ghost" communities to which the process's local communities have incident (inter-community) edges. Given the static nature of input loading, each process knows the vertex and community intervals owned by every other process as well. However, the information pertaining to those vertices and communities could change dynamically and therefore need to be communicated. We use p to denote the number of processes, and rank i to denote an arbitrary rank in the interval $[0, p-1]$.

We also explore an alternative vertex distribution to balance the edge distributions to reduce the number of "ghost" vertices and communication. This distribution maintains roughly equivalent edge lists but may result in an uneven number of vertices per process (and additional file I/O). The impact of this "edge-balanced" distribution in avoiding communication is discussed in the context of distributed-memory evaluation in Section 5.2.

3.2. Overview of the parallel algorithm

The Louvain algorithm has multiple phases, where each phase is run for a certain number of iterations based on a user-defined threshold value. Initially, each vertex is assigned to its own community, and as the algorithm progresses, vertices migrate by entering and leaving different communities. Each vertex resides in one community at the start of an iteration, and decides to either stay in the current community or move to one of its neighboring communities based on the value of modularity gain by the end of an iteration. Algorithm 1 shows a high-level description of the parallel Louvain algorithm executed on process i. In this pseudocode, each iteration of the `while` loop corresponds to a Louvain "phase".

Algorithm 1 shows the two major steps of the parallel Louvain algorithm. The first step involves invoking the Louvain iteration, Line 4 through a call to Function LouvainIteration()), which runs the Louvain heuristic for modularity maximization. The second step is graph reconstruction, Line 5 through a call to Function BuildNextPhaseGraph(), where vertices in each cluster are coarsened into a single meta-vertex, compacting the graph. Function LouvainIteration represents the most compute intensive part of the algorithm, and therefore, benefits from the offloading the computation to a GPU. However, Function BuildNextPhaseGraph involves irregular accesses to memory and can be performed efficiently only on CPUs. We describe these two steps in detail in the following discussion.

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 typically involves information on "ghost" vertices and/or communities. For each vertex owned locally, a community ID is stored; and for each community owned locally, its incident degree (a_c) and weights are stored locally (as part of the vector C_{curr} in Algorithm 2). In addition, each process stores the list of its ghost vertices and their corresponding remote owner processes. Since this vertex mapping to the process space changes with every phase (due to graph compaction), we perform a single (one-time per phase) send-receive communication step to exchange these ghost coordinate information (shown in line 2 of Algorithm 2. Note that the *initial* ghost community information can be derived from the ghost vertex information, as at the start of every phase, each vertex resides

Algorithm 1: Parallel Louvain Algorithm (at rank i).
Input: Local portion $G_i(V_i, E_i, \omega_i$ (in CSR format),
Input: Threshold, τ (default: 10^{-6}), $minSize$
Output: Community assignment C_{curr}

1: $C_{curr} \leftarrow \{\{u\} | \forall u \in V\}$ {Initial community assignment}
2: $G_i^0 \leftarrow G_i$ {G_i^k: Subgraph on rank i and coarsening level k}
3: **while** `true` **do**
4: LouvainIteration($G_i^k, C_{curr,\tau}$) {GPU enabled}
5: $G_i^{k+1} \leftarrow$ BuildNextPhaseGraph(G_i^k, C_{curr}) {On CPUs}
6: **if** $|V(G_i^{k+1})| \leq minSize$ **then**
7: `break`
8: **else**
9: $C_{curr} \leftarrow$ Update based on G_i^{k+1}
10: $G_i^k \leftarrow G_i^{k+1}$ {Update the graph}
11: **return** C_{curr}

Algorithm 2: Algorithm for the Louvain iterations of a phase (rank i, coarse-level k), on CPU and GPU
Input: Local portion $G_i^k(V_i, E_i, \omega_i)$, $\tau = 10^{-6}$
Output: Updates to community assignment C_{curr}

1: **function** LouvainIteration(G_i^k, C_{curr}, τ)
2: $V_g \leftarrow$ Exchange *ghost* vertices
3: {GPU initialization}
4: Declare map $m(V_p) \in \{C_p, a_C\}$ where $C_p \leftarrow C_{curr} \cup C_{remote}$ and $V_p \leftarrow V_i \cup V_g$
5: **while** `true` **do**
6: `send/receive` latest information on all ghost vertices and *update* local communities
7: Remapping: initialize $m(V_p)$
8: Copy $m(V_p)$ from host to device
9: **for** $v \in V_i$ **do** {Computation on GPU}
10: Compute ΔQ by moving v to each of its neighboring communities
11: Determine target community for v based on the migration that maximizes ΔQ
12: Mark both the local and remote communities of v for an update
13: Copy $m(V_p)$ from device to host
14: `send` updated information on ghost communities to owner processes
15: $C_{curr} \leftarrow$ `receive` and update information on local communities
16: $currMod_i \leftarrow$ Compute modularity based on G_i^k and C_{curr}
17: $currMod \leftarrow$ `all-reduce`: $\sum_{\forall i} currMod_i$
18: **if** $currMod - prevMod \leq \tau$ **then**
19: `break`
20: $prevMod \leftarrow currMod$

in its own community. However, 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.

The communities for ghost vertices are maintained in a separate data structure (because they have to be communicated) and its size changes across the iterations (because vertices can move around the communities). To prevent repetitive device allocations of the community data structures across Louvain iterations and to enable coalesced accesses from GPU threads, we combine the local and ghost community data structures and flatten them into separate contiguous sequence containers. A one-time allocation of communities is possible since the number of communities cannot exceed the number of vertices, and in subsequent phases the overall number of communities is expected to shrink; therefore, at most, the per-process size of communities will be equivalent to the number of local and ghost vertices.

The main body of each Louvain iteration consists of the following major steps (see Algorithm 2):

(i) At the beginning of each iteration, obtain information on ghost vertices (i.e., their latest community assignments) at each process (line 2);
(ii) Declare and initialize data structures for device (line 4);
(iii) At the beginning of each iteration, exchange updated information for ghost communities among processes, compute the new community assignments for local vertices (line 6), and adjust GPU data structures (line 7–8);
(iv) Compute the gain in modularity (i.e., ΔQ) if a vertex migrates to a neighboring community, and designate a *target* that maximizes the relative modularity gain (lines 10–11);
(v) Update local and ghost communities, and copy data from device before exchanging the updated communities (lines 10–15);
(vi) Compute the global modularity based on the new community state (line 16–17);
(vii) If the net modularity gain (ΔQ) achieved relative to the previous iteration is below the desired threshold τ, then terminate the phase, and continue otherwise (lines 18–20).

3.3. Graph reconstruction

The coarsening or merging of communities at the input graph level (finest level) can be observed as an important optimization technique to improve the quality of clustering. The communities at the end of the current phase form the basis for the basis for building the coarsened graph for the next phase. Each community is represented as a vertex in the coarsened graph and a self-edge is added with the total number of intra-cluster edges as the weight of this self-edge. The self-edge acts as the balancing force to keep a given cluster in the current phase stay in its own community depending on the number of intra-cluster edges relative to inter-cluster edges. Edges between communities, which are now simply edges between vertices, are assigned weights based on the inter-cluster edges from the current graph.

The graph reconstruction phase in a distributed setting is illustrated using a simple example in Fig. 3. Process #0 owns vertices $\{0, 1, 2\}$, while process #1 owns vertices $\{3, 4\}$. The figure shows the partitioning of the CSR representation. The index array employs local indexes, whereas the edges array has global vertex IDs. Each process has an array identifying community IDs for local vertices, and a hash map that associates remote neighbor vertices with their respective community ID. The specific steps in graph construction are as follows:

(i) Each process counts its unique local clusters, which are renumbered starting from 0. Renumbering is performed using an `std::map` data structure that associates the old community ID with the new ID.

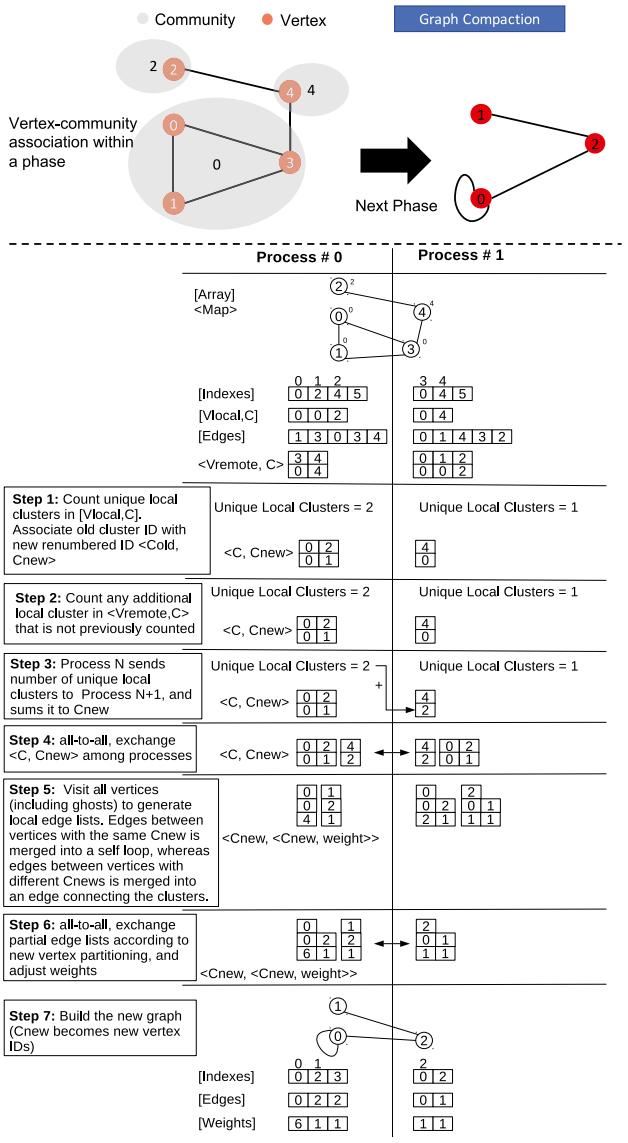


Fig. 3. Graph reconstruction, the *top* cartoon demonstrates the overall compaction process, whereas the *bottom* figure provides more details on the stepwise renumbering and graph CSR reconstruction. In the example, we suppose that the modularity optimization has assigned vertices $\{0, 1, 3\}$ to community 0, vertex 2 to community 2 and vertex 4 to community 4 (i.e., vertices 2 and 4 are each one in their own community). Because community IDs originate from vertex IDs, we consider the community IDs from 0 to 2 owned (local) to process #0, and community IDs 3 and 4 local to process #1.

(ii) Each process checks for local community IDs that, during the Louvain iterations, may have been assigned to remote vertices but are no longer associated with any of the vertices in the local partition.
(iii) Local unique clusters are renumbered globally. This is achieved using a parallel prefix sum computation on the number of unique clusters.
(iv) Processes are involved in communicating the new global community IDs for the local partition. Only the new community IDs that replace the old community IDs used in other processes need to be communicated.
(v) Every process examines each of the vertices in its partition and starts creating partial (new) edge lists. For each vertex in the partition, a process checks its neighbor list. Neighbors associated with the same new community ID contribute to a "self loop" edge.

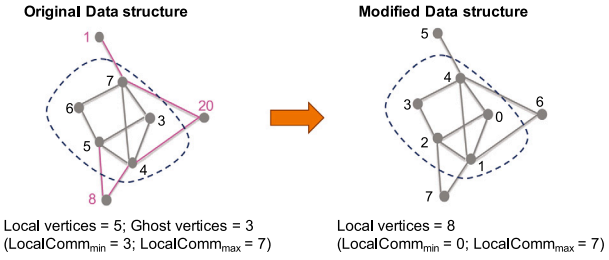


Fig. 4. Vertex index transformation from CPU to GPU representation.

(vi) Once these new partial edge lists have been created, they are redistributed across processes. New partitions are generated so that every process owns an equal number of vertices (as much as possible).
(vii) New arrays for indices and vertices of the coarsened graph can thus finally be rebuilt from the edge lists.

As can be observed from the details, the coarsening function involves simple integer operations and communication between processes. Given the low amount of parallelism in execution, and the need for frequent synchronization, we perform the graph coarsening step only on the CPU side of a node.

4. GPU-porting strategy

As described in the previous sections, the Louvain method comprises two phases. Our distributed heterogeneous multi-GPU implementation currently exploits GPUs only for the modularity optimization phase. The modularity optimization phase can concurrently run on the GPUs and the host CPUs for different portions of the graph. However, the graph coarsening and rebuilding phase is executed only on the host CPUs. In the rest of this section, we discuss critical aspects that enable efficient execution of the algorithm on multiple GPUs across multiple nodes as well as improve execution of the modularity optimization phase on the Volta V100 GPU architecture.

4.1. Mapping between host and device data structures

Similar to Vite, the heterogeneous implementation employs a mix of MPI processes and OpenMP threads. We assign one MPI process per GPU. The graph is partitioned across MPI processes employing a simple 1D vertex based distribution (partitioning). As previously described in Section 3, there is a notion of "ghost" vertices in the graph distribution across processes. In Vite, separate data structures are maintained for vertex-to-community associations to distinguish local (owned by the current process) and ghost vertices. The community associations of the ghost vertices are exchanged with the processes owning them in every iteration. However, the GPU code only performs the modularity optimization part of the algorithm. Hence, once the computation is offloaded to the GPUs, GPUs do not need to communicate data and storing two distinct data structures in GPU memory becomes unnecessary. We, instead, maintain a mapping to distinguish between local and ghost vertices in the single GPU data structure to access the vertex-to-community information for ghost vertices. The size of the mapping is twice the number of edges. An illustration of the index re-mapping using a small example is shown in Fig. 4.

Intuitively, the approach identifies the ghost vertices (which may have any of the global vertex indices) and rebuilds the local CSR representation as if they were local vertex indices. This allows accessing the community information (community assignment and weight of such a community) through direct indexing in an array rather than accessing the separated per-process hash map containing the remote community information. In Fig. 4, the original data structure starts from vertex index 3, as it uses global vertex indices, and gets remapped to a new local CSR starting from local vertex index 0.

The remapping operation to transform the respective per-process vertex indices from the CPU to the GPU representation is currently implemented as a shared-memory parallel routine on the host using OpenMP, and the transformed information is copied to the device. It is possible to entirely eliminate this step by modifying the underlying graph data structure in Vite. However, this would require significant changes to the code, and is therefore a planned set of future optimization.

In Vite, the graph coarsening phase involves reconstructing the Compressed Sparse Row (CSR) representation, and requires re-allocating the data structures as needed. Since GPU memory allocations are expensive, we only allocate a reusable buffer on the device memory once at the beginning, and only adjust pointers to the data structures as required. We also employ pinned memory to accelerate streaming of data from the host process to the device.

4.2. Determining target communities

One of the most expensive operation in a multi-GPU implementation is identifying target communities for vertices. In a Louvain iteration, vertices calculate the relative gain in modularity obtained by moving the vertex from its current assignment to one the neighboring communities. In the CPU-only version, the community information (size and degree of communities) is stored as a C++ STL datastructure `std::map`, and STL functions are used to "find" if a specific community already exists in the map. GPU implementations of C++ containers are not portable across CUDA releases (for e.g., CUDA Thrust libraries [31]), and the third-party libraries also suffer from similar limitations. Furthermore, the absence of a scalable hash function complicates the search in the data dictionary on a GPU. We are aware of the existing research on faster hash functions targeted for GPUs [32–34]. We plan to employ such techniques in our future work.

In the current implementation, we maintain two independent vectors to store degree and sizes. The task of determining the target community for a vertex is either undertaken by a tile or a block (multiple tiles) of threads. This is decided on the base of the degree of a given vertex. If the degree of a vertex is on the higher side, then a block of threads is dispatched to handle this operation, else a tile is sufficient. In a tile or a block, each thread processes one or more edges of a vertex to identify the neighboring communities and compute the changes in modularity for moving the vertex being processed to one of them. Additionally, in order to mitigate load imbalances owing to varied vertex degree distributions, we make use of two separate CUDA streams per GPU to distinguish computations on high-degree vertices from the rest.

4.3. Exploiting CUDA cooperative groups

CUDA cooperative groups (CGs) introduced with CUDA 10.0, provide a flexible model for synchronization and communication within groups of threads [35,36] of arbitrary dimensions, differently from the basic thread block based synchronization historically employed in previous CUDA versions. The cooperative groups programming model leverages four key elements: (a) Group partitioning; (b) collective algorithms for data movement and manipulation; (c) group barrier synchronization; and, (d) collectives that expose low-level group-specific operations. While cooperative groups are supported on older GPU architectures such as Pascal, the new execution model for warps in Volta, which significantly reduces penalties for thread divergence, makes it an ideal platform for implementing cooperative groups. We made extensive use of CGs for implementing all the GPU kernels in cuVite.

Fig. 5 illustrates the template for using CGs in cuVite. The degree of a vertex is used to decide the size of a cooperative group that will be assigned to process a given vertex. If the degree is higher than a

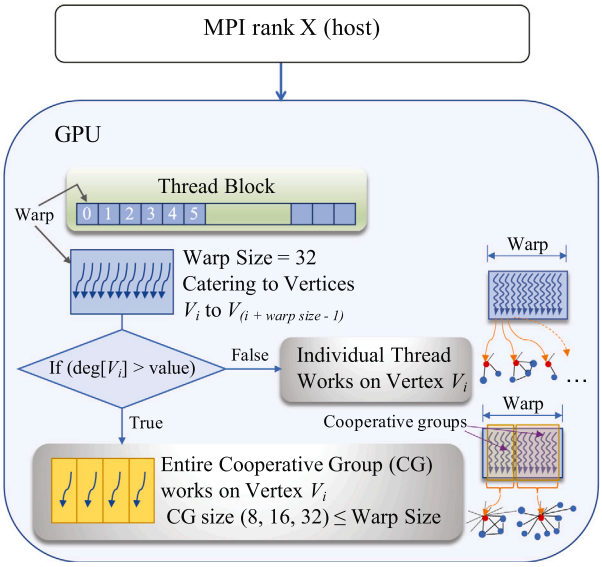


Fig. 5. Vertex distribution among GPU threads: depending on the vertex degrees, individual threads in a warp work on a vertex or an entire warp works on a single vertex, requiring cooperative groups for synchronization.

predefined value (the tile size for a CG), we assign an entire cooperative group to work on a single vertex, since it is easy to synchronize once execution is completed for this vertex. When the size of a CG equals the size of a warp, we observe the highest performance. However, on Volta, CGs enable the use of smaller number of threads without significant penalties. CGs are also easy to synchronize at the subwarp level. When the degree of a vertex is very small, we assign a single thread per vertex. This approach is particularly useful during modularity optimization, when each vertex needs to identify the communities of its neighborhood and evaluate the variation in modularity when moving to each one of them in order to select the move that provides the highest increase.

In the following section (Section 5), we present the experimental results detailing the impact of the data structures modifications and of the other optimization techniques discussed here.

5. Performance evaluation

In this section, we discuss our distributed-memory Louvain evaluations on ALCF Theta and OLCF Summit platforms (refer to Section 2.1), using both real-world and synthetic graphs (we use *undirected* representation of graphs). We begin by characterizing some general observations of our distributed-memory implementation in Section 5.1. We discuss our evaluations on ALCF Theta in the context of exploiting the modes to access KNL MCDRAM in Section 5.2. Finally, in Section 5.3, we discuss our distributed-memory GPU evaluations on OLCF Summit.

5.1. General performance characterization

Overall performance of our distributed implementation is sensitive to the input graph (especially since our simple graph partitioning makes no assumption about the underlying graph structure). Fig. 6 shows the inter-process communication volume of distributed Louvain method for four real-world graphs on 1024 processes of Theta respectively, and they exhibit significantly different communication patterns.

Load balancing of real-world graphs is challenging, since it is non-trivial 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 extra I/O to read the graph. We embed the edge count per vertex information in an intermediate binary CSR representation of the native graph file (refer to Section 3.1), in addition to the edge list. A single process can read the graph partially within a limited amount of time (since number of vertices is usually significantly lesser than the number of edges) and use the per-vertex edge count information to construct a partitioning scheme that tries to balance the number of edges owned by a process, and accordingly broadcasts the respective file read positions to rest of the processes.

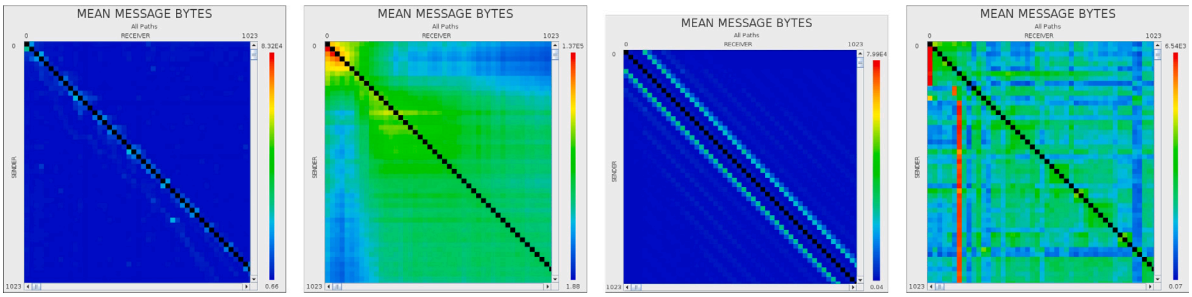
Fig. 8 demonstrates the standard deviation of edges owned by a process in the classic vertex-based distribution that divides the number of vertices among processes (with each process receiving all the edges connected to a vertex, in addition to "ghost" vertices), as compared to our edge-balanced distribution, for eight real-world graphs distributed on four process configurations (256, 512, 1024 and 2048). Thus, the edge-balanced distribution can significantly minimize the amount of "ghost" vertices, leading to communication avoidance. As a result, we observe up to 80% improvement in the end-to-end execution times for clustering, compared to the standard distribution, for a number of real-world cases.

We use MPI nonblocking Send/Recv and collectives to perform communication in our distributed Louvain implementation. Exchanging vertex-community association among processes take place in every iteration of a phase (refer to Algorithm 2), and is the most expensive communication operation. In general, communication overhead of our implementation can be significant and more than 90% of the overall elapsed time as shown in Fig. 7, depending on the input graph and process configurations, primarily due to the inherent load imbalances in this application. Fig. 7 also shows the memory per PE (as reported by CrayPAT profiler), which is an artifact of the distributed graph structure. Hence, we include maximum, average and standard deviation of edge distribution across the PEs. Relatively high standard deviation of edge distribution indicates a higher number of "ghost" vertices.

5.2. Performance on ALCF Theta

The KNL nodes of ALCF Theta allows multiple configuration for exploiting the on-package MCDRAM. In this section, we analyze the impact of the *memory modes* or MCDRAM configurations on performance for our distributed Louvain implementation. It is possible to configure the available memory in KNL nodes of Theta into one of the three modes—(i) *cache*: MCDRAM is a cache for main memory; (ii) *flat*: MCDRAM is treated as an addressable memory (like main memory); and, (iii) *hybrid*: a portion of MCDRAM is treated as addressable memory, and the rest is a cache for main memory. We further classify *hybrid* mode into *equal* and *split*. In *equal* memory mode, 50% of MCDRAM is addressable memory, and the other 50% is a cache. Whereas, in *split* mode, 75% of MCDRAM is addressable memory, and the remaining 25% is cache. We use a custom allocator (i.e., `hbw::allocator`) from the `memkind` library [37] to allocate C++ containers that store the community size/degree and the vertex-community mapping on the KNL MCDRAM. The performance differences between different clustering modes were not evident, therefore we selected the default *quadrant* for our distributed Louvain implementation. In *quadrant* clustering mode, the tiles are divided into four parts (quadrants), which are spatially located near four groups of memory controllers. Keeping the clustering mode constant, we vary the memory modes and demonstrate performance using four real-world datasets in Fig. 9.

Despite of the inherent simplicity of the *cache* mode (no application code modification), the access latency of MCDRAM is higher than standard caches, and the overall memory bandwidth is impacted by main memory accesses (for the portion of data not resident on the MCDRAM). Due to the irregular nature of memory accesses in our distributed Louvain implementation, cache misses are pervasive. In *cache* mode, the MCDRAM in KNL is treated as a direct mapped cache, in which an address in the main memory is mapped to only one location in the cache. Whereas, L3 caches in conventional CPU architectures



(a) uk-2007 (b) com-friendster (c) nlpkkt240 (d) com-orkut

Fig. 6. Communication volume, in terms of mean `send/recv` message sizes (bytes) exchanged between pairs of processes, for two real-world inputs on 1024 processes. The vertical axis represents the sender process ids and the horizontal axis represents the receiver process IDs.

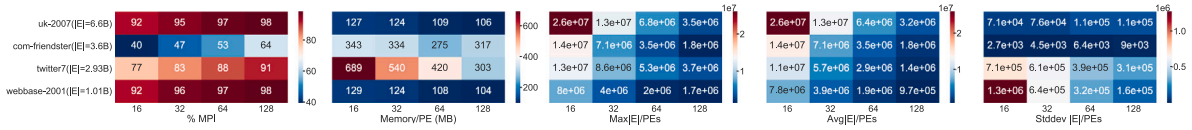


Fig. 7. % MPI, Memory/PE and edge distribution across PEs for various real-world graphs on 16–128 Theta nodes.

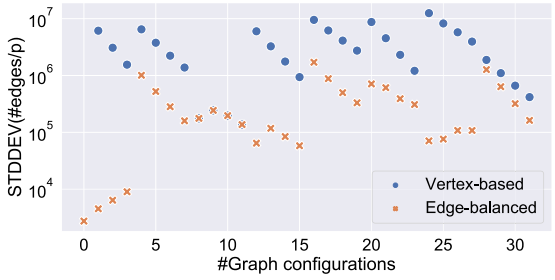


Fig. 8. Lower standard deviation of #Edges/process of the edge-balanced distribution compared to classic vertex-based distribution indicates less communication for the former.

Table 1
Graphs used in summit multi-GPU evaluations.

Graphs	$\|V\|$	$\|E\|$	Modularity	Iterations	Phases
Synthetic graph datasets					
rgg-33M	33.55M	378.02M	0.99	77	8
rgg-67M	67.10M	775.04M	0.99	51	4
rgg-134M	134.22M	1.58B	0.99	93	9
Real-world datasets					
hollywood-2009	1.14M	113.89M	0.75	82	9
nlpkkt240	27.99M	760.64M	0.97	714	8
uk-2002	18.52M	298.11M	0.99	43	5
uk-2005	39.46M	936.36M	0.95	79	12
webbase-2001	118.1M	1.01B	0.98	47	7
com-friendster	65.61M	3.6B	0.61	39	3
uk-2007	105.90M	6.6B	0.99	32	5

such as Intel® Haswell® are multi-way set associative, in which an address in main memory can be mapped to any of the multiple cache addresses, significantly reducing conflict misses. An MCDRAM cache miss is more expensive than reading from main memory, because memory requests cannot travel from processor L2 cache to main memory directly, and has to involve MCDRAM in between. We notice that the hybrid *split* mode is more scalable than the other modes, whereas the *flat* mode (opposite of the *cache* mode) yields the best execution time performance in most of the cases. In *flat* mode, we explicitly allocated some data structures on the MCDRAM, and observe 30%–45% better performance as compared to the *cache* mode. We capture the relative performances between the KNL memory modes for our distributed Louvain implementation in Fig. 10.

5.3. Performance on OLCF Summit

We now present multi-GPU results on OLCF Summit. The input consists of real-world graphs obtained from the SuiteSparse Matrix Collection [38] and IEEE-HPEC Graph Challenge [39], and synthetically generated using the random geometric graph (RGG) model [12], as listed in Table 1. Several heuristics such as threshold scaling and incomplete coloring exist in Vite for significantly improving the performance at scale [11]. However, our goal in this work is to conduct a baseline performance analysis, and therefore, we refrained from using heuristics in evaluations. On Summit we use GCC 8.1 compiler, CUDA 10.1.243 and Spectrum MPI 10.3 for building cuVite.

In Fig. 11, we compare cuVite results on multiple GPUs across Summit nodes, using similar runs of Vite as a baseline for comparison. For these evaluations, the only difference between Vite and cuVite configuration is in the usage of a GPU per MPI rank, while the number of threads per rank is kept the same for both of these versions. As we discussed in Section 4.1, the GPU implementation has to undergo a remapping operation on the host side, which can take more than 2× the time spent in GPU computation for certain large graphs, such as com-friendster (due to the presence of many high-degree vertices). While remapping is a costly operation with respect to the GPU computation, the adopted solution allows quickly designing a hybrid solution starting from the Vite code and data structures, optimized for CPU, without completely redesigning the application. One objective of this work is, in fact, not only showing a multi-GPU porting, but a hybrid implementation, where general-purpose processing elements are also used to provide further scalability.

Barring the remapping costs, we observed 1.2–3.3× improvement in performance of the GPU version for most of the input graphs with respect to the CPU-only Vite. cuVite implementation demonstrates about 1.6–3.2× scalability over 2–64 Summit nodes for most of the graphs in Fig. 11. Since graph edge distribution varies with the number of processes, an irregular number of edges per process can cause severe load imbalance, limiting the overall scalability.

5.4. Performance on NVIDIA DGX-2

We now compare the performance of cuVite relative to NVIDIA RAPIDS® cuGraph [40], Rundemanen [27] and Grappolo [26]. The GPU

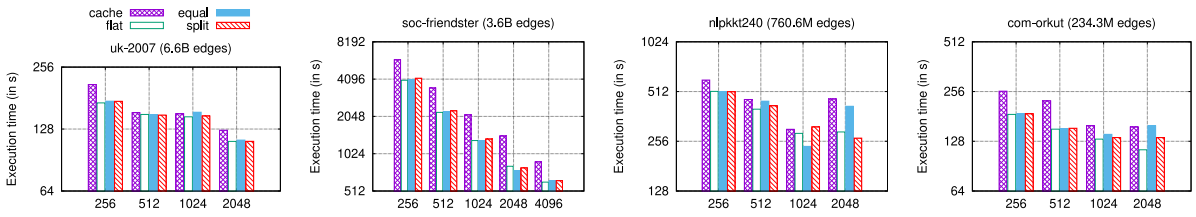


Fig. 9. Performance of real-world graphs using different memory modes on KNL nodes of ALCF Theta. *X-axis*: #Processes; *Y-axis*: Execution time (s).

Table 2
Single node performance of cuVite relative to NVIDIA RAPIDS® cuGraph and Rundemanen on NVIDIA V100 GPU and Grappolo (using 224 OpenMP threads). An asterisk is shown for runs that did not complete because of insufficient memory to execute Rundemanen. Highlighted rows signify cases for which cuVite provided the best performance relative to others.

Graphs			cuGraph		Rundemanen		cuVite		Grappolo (CPU; 64-bit)	
Name	$\|V\|$	$\|E\|$	Modularity	Time (s)	Modularity	Time (s)	Modularity	Time (s)	Modularity	Time (s)
as20000102	6.47K	50.2K	0.62	0.04	0.61	0.03	0.61	1.02	0.61	0.05
Oregon2_010512	11.26K	62.6K	0.63	0.04	0.63	0.14	0.63	0.12	0.63	0.10
p2p-Gnutella31	62.58K	295.78K	0.48	0.12	0.47	0.21	0.47	0.25	0.47	0.11
soc-Epinions1	75.87K	811K	0.45	0.12	0.43	0.34	0.43	0.13	0.43	0.51
soc-Slashdot0902	82.16K	1M	0.33	0.16	0.29	0.24	0.37	0.41	0.26	0.66
flickrEdges	105.93K	4.63K	0.67	0.39	0.67	0.43	0.67	0.43	0.67	8.91
roadNet-PA	1.08M	3.08M	0.98	0.76	0.98	1.02	0.98	0.32	0.98	1.80
roadNet-TX	1.37M	3.84M	0.99	0.88	0.99	1.22	0.99	0.39	0.99	1.69
roadNet-CA	1.96M	5.53M	0.99	1.05	0.99	1.66	0.99	0.62	0.99	5.04
V2a	55.04M	117.21M	0.99	5.72	-	*	0.99	30.29	0.99	29.12
U1a	67.16M	138.77M	0.98	4.65	-	*	0.98	27.42	0.98	77.29
Graph500-scale21	1.24M	63.4M	0.06	1.45	0.05	3.52	0.02	4.17	0.04	37.04
Graph500-scale22	2.39M	128.19M	0.07	3.39	0.05	7.17	0.02	8.89	0.03	46.91
Graph500-scale23	4.60M	258.5M	0.06	5.82	0.05	14.04	0.02	22.03	0.03	174.59
MAWI-1	18.57M	38.04M	0.25	59.02	0.89	14.08	0.89	6.97	0.88	68.79
MAWI-2	35.99M	74.48M	0.25	297.74	-	*	0.9	16.52	0.88	95.25
MAWI-3	68.86M	143.41M	0.28	520.77	-	*	0.89	28.1	0.87	323.94
MAWI-4	128.56M	270.23M	0.21	1533.17	-	*	0.89	78.85	0.86	498.17

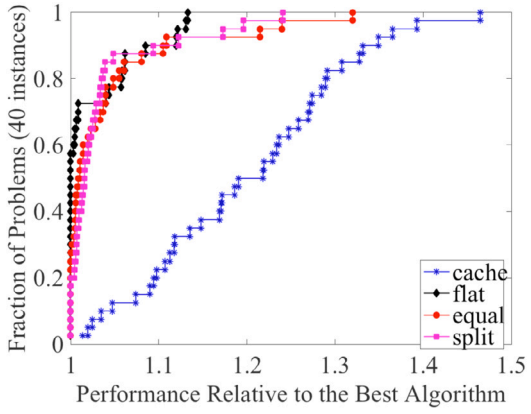


Fig. 10. The relative performance profiles for *cache*, *equal*, *flat* and *split* memory modes on Theta KNL nodes using all the runs as shown in Fig. 9 in addition to similar runs with other graphs not included. The X-axis represents the factor by which a given scheme fares relative to the best performing scheme for that particular input. The Y-axis represents the fraction of problems. The closer a curve is aligned to the Y-axis the superior is its performance relative to the other schemes over a range of 40 inputs.

implementations were executed on a single-GPU of NVIDIA DGX-2 platform. We use GCC 7.3, CUDA 10.0.130 and OpenMPI 3.1.3 to build our GPU implementations. Comparative Grappolo evaluations was performed on a 224-core 8-way (28-core/socket) Intel® Xeon® Platinum 8276M CPU 2.20 GHz system, with 1MB private L2 cache and shared 38.5MB L3 cache. We used GCC/9.2.0 compiler to build Grappolo on the shared-memory platform.

From an algorithmic perspective, cuVite and Rundemanen are similar in that they both adapt parallelization strategies from Grappolo. The latest GPU-based implementation is in the NVIDIA RAPIDS software suite [41]. We used cuGraph 0.14.0 in our experiments and built the Python dependencies using Anaconda release 3.2019.3. We only time the cugraph. `community.louvain` function,[2] while comparing the performance and modularity scores with our GPU implementation on a single GPU of the NVIDIA DGX-2 system.

As summarized in Table 2 the results computed from our GPU implementation are comparable with `Grappolo` for most of the graphs, which has been shown to closely resemble the output from the Louvain implementation of Blondel et al. [21]. Since cuGraph currently does not support multiple GPUs, we use a single V100 GPU for comparison. We also used a single MPI rank and 8 OpenMP threads for cuVite evaluation. Since there are no "ghost" vertices in this case, there is no remapping overhead as well (refer to Section 4.1). We selected a number of real-world graphs and three synthetic ones (for e.g., Graph500); unlike the real-world instances, the Graph500 inputs exhibit poor community structure, as evident from the low modularity scores.

Since cuGraph uses a 32-bit representation to store graph vertex indices and edge weights, cuVite also used a 32-bit representation for this evaluation, along with Rundemanen. However, we use 64-bit representation of Grappolo as there no option to build the software with 32-bit data representation. For all other cases, we have used a default 64-bit representation for cuVite to store the graph and for associated operations. We also observe that modularity values for cuVite are comparable to Grappolo.

As shown in Table 2, cuVite demonstrates a speedup of about 2–19× relative to cuGraph for a number of cases. Since it is not apparent how to extract information for each Louvain phase in cuGraph, we are unable to analyze the reason behind modularity divergences for few of the inputs,

[2] https://docs.rapids.ai/api/cugraph/stable/api.html#module-cugraph.community.louvain

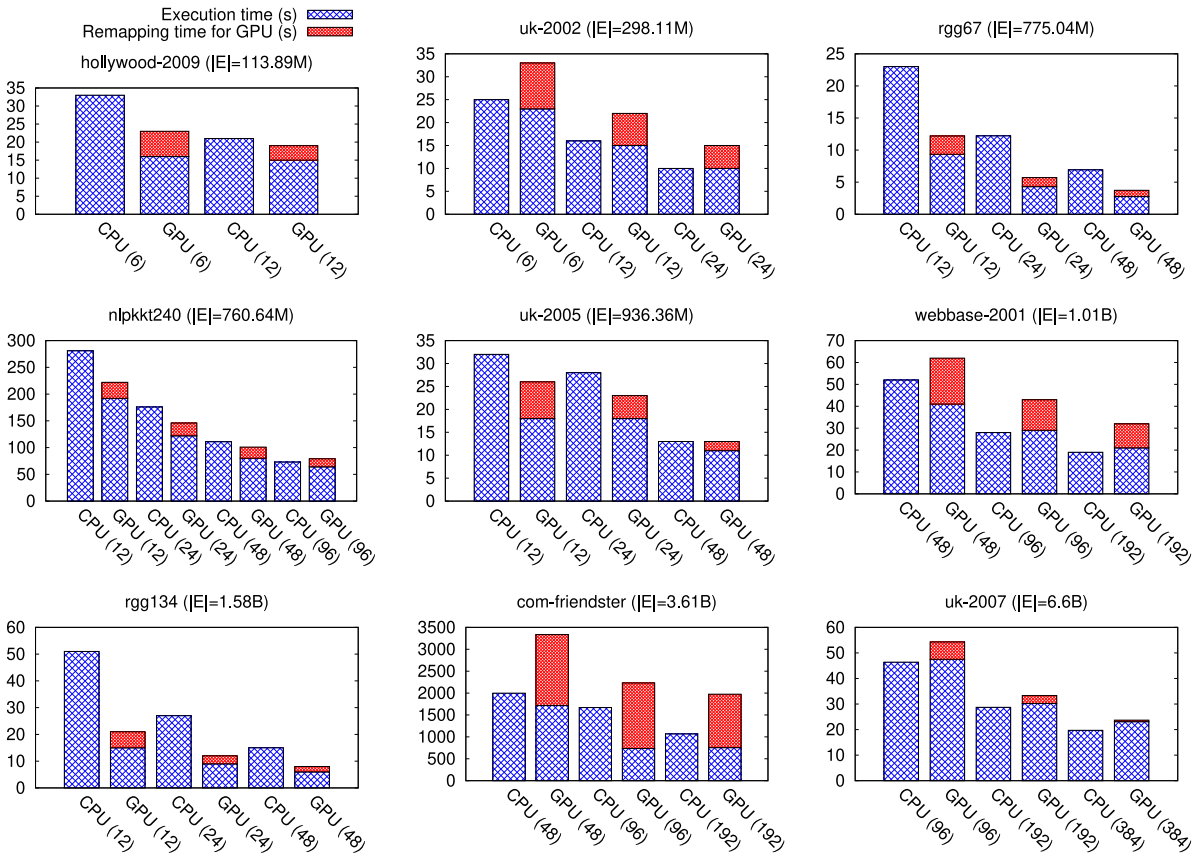


Fig. 11. *Strong scaling of* Vite *and* cuVite on OLCF Summit using real-world and synthetic graphs. **X-axis**: CPU/GPU version on total #processes (with 6 MPI ranks per node, 14 OpenMP threads per MPI rank; additionally, GPU versions use a single GPU per MPI rank); **Y-axis**: Time in seconds.

such as MAWI. Unlike GPU implementations of cuGraph and Rundemanen where nearly all computations over multiple phases occur in the GPU memory, in our implementation the community updates are performed by the CPU. Therefore, in our current implementation, the data on community assignments and their weights need to be transferred back and forth between host and device in every iteration. This cost of moving data between host and device can be a significant fraction of the total time for smaller datasets, for which cuGraph and Rundemanen perform better than cuVite. Also, the current version of cuGraph Louvain requires the difference between the current and previously computed global modularities to be at least 1.0E−03,[3] whereas for Vite/cuVite/Grappolo it is 1.0E−06. This can cause a mismatch in the number of iterations to convergence for cuVite and cuGraph.

5.5. Quality assessment

In order to assess the quality of our distributed-memory multi-GPU Louvain implementation, we compare our results against communities reported by the CPU implementation, i.e., Vite. We use standard statistical measures such as F-score, Precision and Recall that are computed from the true positive, false positive and false negative evaluations obtained by comparing all possible vertex pairs in community assignments reported by Vite and cuVite. As summarized in Table 5, values close to 1 indicate an almost exact clustering of vertices by cuVite relative to Vite. Further details are also presented in Table 2. We present results on qualitative and performance comparisons of cuVite relative to Vite and Grappolo in this section.

[3] https://github.com/rapidsai/cugraph/blob/branch-21.10/cpp/src/community/louvain.cuh#L365

Table 3
Comparing Grappolo with Vite (distributed CPU) and cuVite (distributed CPU+GPU, this work) for large graphs.

Graphs	Grappolo (128 threads)	Vite (32 nodes)	cuVite (32 nodes)	
			Remap	GPU
nlpkkt240	283	73	15	64
uk-2005	20	13	2	11
webbase-2001	43	19	11	21
com-friendster	1824	1068	1220	755

Table 4
Comparing single process execution times (in seconds) with multiple processes (and GPUs) for two medium graphs.

PEs (1 node)	rgg33 (|E|=378.02M)		Hollywood-2009 (|E|=113.89M)	
	Vite	CuVite	Vite	CuVite
1	171	–	36	19
2	76	20	52	26
4	38	13	29	25
8	43	25	33	23
12	28	16	21	19

Table 3 compares the GPU results for few of the larger graphs with Grappolo on 128 threads. Single process or serial execution times for most of these graphs are quite high, as noted by Lu et al. [26]. In Table 4, we compare Vite and CuVite execution times using two structurally different medium-sized graphs over 1–12 processes. While rgg33 demonstrates up to 6× scalability on 12 PEs, hollywood-2009 only exhibits about 40% improvement in the execution time.

We note that Ghosh et al. performed quality assessment of Vite [11] relative to the original serial implementation by Blondel *et al.* In Table 6, we use several synthetic graphs from the LFR benchmark [42]

Table 5
Quality assessment of cuVite relative to Vite for four real-world inputs. Values close or equal to one indicate strong correlation with CPU results.

Score	Hollywood-2009	uk-2002	nlpkkt240	webbase-2001
Precision	0.985	0.995	1.0	1.0
Recall	0.956	0.933	1.0	1.0
F-score	0.970	0.963	1.0	1.0

Table 6
Quality assessment of Vite across multiple processes for various LFR graphs, and comparison with Grappolo (multi-threaded).

$\|V\|$	$\|E\|$	#PEs	Time(s)	$F-score_{\text{Vite}}$	$F-score_{\text{Grappolo}}$
350K	34.72M	1	114.62	0.990352	0.990352
600K	58.91M	32	112.90	0.990849	0.990849
1M	98.12M	208	117.40	0.981119	0.981119
1.5M	147.13M	448	116.40	0.967736	0.967736
2M	196.45M	512	113.55	0.945176	0.951238

to demonstrate the quality across 1–512 PEs (up to 16 nodes) of Vite vs. @shared-memory Grappolo, compared with ground-truth communities. Overall, we observe about 6%–10% variability in the relative community assignments with larger graphs and #PEs, due to the inherent non-deterministic nature of the algorithm. Since we compare with Grappolo, there is existing empirical analysis using datasets with known ground truth information [43].

6. Related work

Graph analytics has emerged as an important branch of data analytics and enables efficient modeling and analysis of unstructured data with complex relationships among participating entities. Community detection is a commonly used tool in graph analytics to not only discover coherent modules or structures in a graph, but also for dimensionality reduction in many applications. Consequently, community detection has been studied extensively in literature. Our work has therefore benefited from advances in several fronts including superior algorithms, parallelization efforts and application domains. In particular, modularity optimization has been studied extensively and has is being used widely in numerous science domains and analytics applications. We refer you to comprehensive survey papers for details [4–7].

Parallel implementations of community detection that are relevant to this work can be broadly categorized based on the target architectures into: CPU-only, GPU-only, and heterogeneous CPU-and-GPU. Shared-memory or multithreaded CPU-only implementations have been shown to scale and perform well [44–46], leading to the exploration of a number of heuristics and push–pull formulations that prune unnecessary edge explorations [47]. Since multi-threaded implementations are severely restricted on the sizes of inputs that can be processed, we focus only on distributed-memory and GPU-based implementations in the following discussion.

CPU-only Efforts: Distributed homogeneous CPU implementations of community detection for high-performance computing (HPC) clusters, taking advantage of novel network interconnects, system architectures, and specialized algorithm designs, have demonstrated scalability well up to several thousand computing cores distributed over hundreds of computing nodes [11,48–56]. Among these, in the MPI-based distributed memory Louvain implementation of Que et al. [55], the vertices and their edge lists are partitioned among the processes using a 1D decomposition, similar to our distribution strategy. However, our approaches are significantly different. Firstly, we use various heuristics to optimize performance. Moreover, we use large real-world datasets in our experimental evaluations, and compare the performance of our MPI+OpenMP Louvain algorithm with that of a pure OpenMP implementation, and recursively, with the original serial implementation of Blondel et al. Que et al. [55] report the execution time for their algorithm run on the `uk-2007` real-world network (3.3B edges) to be about 45 s on 128 IBM® Power7® nodes. In comparison, we report an all-inclusive execution time of about 61 s for the same uk-2007 graph on 256 Intel® KNL nodes of ALCF Theta, and 20 s on 64 dual-socket IBM Power9 nodes of OLCF Summit (using 4 OpenMP threads per process). Whereas on 64 Summit nodes using 384 GPUs (6 GPUs per node), we obtain a total execution time of 23 s for our GPU version (refer to Fig. 11).

In the MPI implementation of Wickramaarachchi et al. [48], a parallel graph partitioner ParMETIS [57] is used to partition the graph among processes before the distributed memory community detection algorithm starts. Since graph partitioning is also an NP-hard problem and generally more expensive than community detection, we do not assume an optimized input distribution in our approach and instead work with a simpler distribution. It has also been well studied that graphs from real-world are often hard to partition beyond a small number of partitions [58]. In a similar work, Zeng et al. [59] present their distributed-memory (MPI-based) Louvain implementation where they replicate high-degree vertices among processes and redistribute edges to ensure load balancing of edges. The authors report that the execution time of the first two Louvain phases on the uk-2007 graph is over 100 s on 1024 processes of the ORNL Titan supercomputer. In contrast, the execution time of the baseline version of our distributed Louvain implementation including all the Louvain phases for the uk-2007 graph is about 38 s on 1024 processes of NERSC Cori.

GPU-based Efforts: Several works have shown the potential of GPUs to accelerate the Louvain algorithm. While Naim et al. [27] showed that single-GPU implementation was significant faster relative to a CPU-based parallel implementation using a large set of problems [27], the maximum problem size that can be executed in memory of a single GPU remains severely restricted. We provide relative performance of cuVite in Section 5.

Cheong et al. [60] addressed multi-GPU scalability, demonstrating speed up of the multi-GPU implementation with respect to a single-GPU solution, but they started with a slower single-GPU performance relative to the work of Naim et al. [27]. They also showed a degradation in the quality of results. Another multi-GPU parallel Louvain algorithm implemented in the tool suite `Gunrock`, a graph processing library for GPUs, has demonstrated scalability with respect to its specific single-GPU implementation. However, it was evaluated only on clusters with a single GPU per node, and remained limited by the underlying library primitives [61,62]. Most recently, NVIDIA® introduced a software suite named RAPIDS®, which also includes the cuGraph library for optimized graph algorithms such as community detection using the Louvain algorithm [40]. However, cuGraph is a single-GPU code, which we included in our empirical analysis (Section 5).

Heterogeneous CPU-and-GPU implementations have been demonstrated only on single shared-memory CPU nodes. One such example is the work of Bhowmick et al. which focused on the static work distribution among the multi-core CPUs and a single-GPU in the node [63]. We note that the implementation of Bhowmick et al. is based on individual tools used in our analysis (Section 5).

To the best of our knowledge, this work presents the first *distributed-memory multi-GPU* community detection using the Louvain method as the serial template, and made available in a library names cuVite. Although we build on the CPU-only distributed-memory implementation, Vite [11], we made significant additions to exploit multi-GPU nodes.

7. Conclusion

Massively parallel systems including the forthcoming exascale systems are fundamentally heterogeneous in nature comprising of node with multi-core CPUs and multiple GPUs. These hierarchical systems pose significant challenges for the design and development of efficient

parallel algorithms, especially irregular applications such as iterative graph algorithms. In this paper, we presented cuVite, a distributed-memory multi-GPU community detection algorithm using the Louvain algorithm as the serial template, and using the latest CUDA features. We demonstrated the quality and performance of cuVite using a set of real-world and synthetic graphs. We also showed the relative performance of cuVite with the state-of-the-art shared and distributed-memory CPU-only as well as GPU-based implementations. We presented performance improvement of up to 19× relative to NVIDIA cuGraph and greater than 10× relative to multithreaded CPU implementation Grappolo. Our code is available from: https://github.com/pnnl/cuVite.

The performance of cuVite is currently limited by the data transformations needed to map information between host and device data structures, and adds significant overhead in some cases. However, these transformations can be eliminated by using an uniform data representation on both the device and host. We therefore plan to perform a complete redesign of the elementary data structures in the near future, which would also enhance the performance of Vite along with the performance of cuVite.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported by the DOE ECP (17-SC-20-SC), a collaborative effort of the U.S. DOE SC and the NNSA, and by the U.S. NSF grants CCF 1815467 and CCF 1919122 to Washington State University. We used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, United States (Contract DE-AC05-00OR22725) and Argonne Leadership Computing Facility at Argonne National Laboratory, United States (Contract DE-AC02-06CH11357). The Pacific Northwest National Laboratory is operated by Battelle Memorial Institute, United States under Contract DE-AC06-76RL01830.

References

[1] C.C. Aggarwal, Graph clustering, in: C. Sammut, G.I. Webb (Eds.), Encyclopedia of Machine Learning, Springer US, Boston, MA, 2010, pp. 459–467.
[2] M.E. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69 (2) (2004) 026113.
[3] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, D. Wagner, On modularity clustering, IEEE Trans. Knowl. Data Eng. 20 (2) (2007) 172–188.
[4] S. Fortunato, Community detection in graphs, Phys. Rep. 486 (3) (2010) 75–174.
[5] M. Coscia, F. Giannotti, D. Pedreschi, A classification for community discovery methods in complex networks, Stat. Anal. Data Min.: ASA Data Sci. J. 4 (5) (2011) 512–546.
[6] M.E. Newman, Communities, modules and large-scale structure in networks, Nat. Phys. 8 (1) (2012) 25.
[7] M.A. Porter, J.-P. Onnela, P.J. Mucha, Communities in networks, Not. AMS 56 (9) (2009) 1082–1097.
[8] A. Pothen, Graph Partitioning Algorithms with Applications to Scientific Computing, Tech. Rep., Old Dominion University, USA, 1997.
[9] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, Challenges in parallel graph processing, Parallel Process. Lett. 17 (01) (2007) 5–20.
[10] M. Halappanavar, A. Pothen, A. Azad, F. Manne, J. Langguth, A. Khan, Codesign lessons learned from implementing graph matching on multithreaded architectures, Computer 48 (8) (2015) 46–55.
[11] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarria-Miranda, A. Khan, A. Gebremedhin, Distributed louvain algorithm for graph community detection, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2018, pp. 885–895.
[12] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, A.H. Gebremedhin, MiniVite: A graph analytics benchmarking tool for massively parallel systems, in: 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS, IEEE, 2018, pp. 51–56.
[13] S. Jeaugey, Nccl 2.0, in: GPU Technology Conference, Vol. 2, GTC, 2017.
[14] B. Karrer, M.E. Newman, Stochastic blockmodels and community structure in networks, Phys. Rev. E 83 (1) (2011) 016107.
[15] D. Krackhardt, R.N. Stern, Informal networks and organizational crises: An experimental simulation, Soc. Psychol. Q. (1988) 123–140.
[16] Z. Li, S. Zhang, R.-S. Wang, X.-S. Zhang, L. Chen, Quantitative function for community detection, Phys. Rev. E 77 (3) (2008) 036109.
[17] S. Fortunato, M. Barthélemy, Resolution limit in community detection, Proc. Natl. Acad. Sci. 104 (1) (2007) 36–41.
[18] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, D. Wagner, Maximizing modularity is hard, 2006, arXiv preprint Physics/0608255.
[19] B.H. Good, Y.-A. de Montjoye, A. Clauset, Performance of modularity maximization in practical contexts, Phys. Rev. E 81 (4) (2010) 046106.
[20] V.A. Traag, P. Van Dooren, Y. Nesterov, Narrow scope for resolution-limit-free community detection, Phys. Rev. E 84 (1) (2011) 016114.
[21] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, J. Stat. Mech.: Theory Exp. 2008 (10) (2008) P10008.
[22] D. Hric, R.K. Darst, S. Fortunato, Community detection in networks: Structural communities versus ground truth, Phys. Rev. E 90 (6) (2014) 062805.
[23] C.L. Staudt, H. Meyerhenke, Engineering high-performance community detection heuristics for massive graphs, in: Parallel Processing (ICPP), 2013 42nd International Conference On, IEEE, 2013, pp. 180–189.
[24] D. LaSalle, G. Karypis, Multi-threaded modularity based graph clustering using the multilevel paradigm, J. Parallel Distrib. Comput. 76 (2015) 66–80.
[25] E.J. Riedy, H. Meyerhenke, D. Ediger, D.A. Bader, Parallel community detection for massive graphs, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2011, pp. 286–296.
[26] H. Lu, M. Halappanavar, A. Kalyanaraman, Parallel heuristics for scalable community detection, Parallel Comput. 47 (2015) 19–37.
[27] M. Naim, F. Manne, M. Halappanavar, A. Tumeo, Community detection on the GPU, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2017, pp. 625–634.
[28] W. Weir, S. Emmons, R. Gibson, D. Taylor, P. Mucha, Post-processing partitions to identify domains of modularity optimization, Algorithms 10 (3) (2017) 93.
[29] R. Barik, M. Minutoli, M. Halappanavar, N. Tallent, A. Kalyanaraman, Vertex reordering for real-world graphs and applications: An empirical evaluation, in: IEEE International Symposium on Workload Characterization, IISWC'20, Virtual Conference, October 27–29, 2020, IEEE Computer Society, 2020.
[30] J. Dongarra, Compressed Row Storage, http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html.
[31] N. Bell, J. Hoberock, Thrust: A productivity-oriented library for CUDA, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.
[32] D.A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J.D. Owens, N. Amenta, Real-time parallel hashing on the GPU, in: ACM SIGGRAPH Asia 2009 Papers, 2009, pp. 1–9.
[33] D.A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J.D. Owens, N. Amenta, Building an efficient hash table on the GPU, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 39–53.
[34] S. Ashkiani, S. Li, M. Farach-Colton, N. Amenta, J.D. Owens, GPU LSM: A dynamic dictionary data structure for the GPU, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2018, pp. 430–440.
[35] K. Perelygin, Y. Lin, Cooperative groups, in GTC, 2017, NVIDIA.
[36] NVIDIA, CUDA Toolkit Documentation, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[37] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, S.D. Hammond, memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies, Tech. Rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States, 2015.
[38] S.P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T.A. Davis, M. Henderson, Y. Hu, R. Sandstrom, The suitesparse matrix collection website interface, J. Open Source Softw. 4 (35) (2019) 1244.
[39] MIT, A. Webservices, Graphchallenge, 2020, http://graphchallenge.mit.edu/data-sets.
[40] NVIDIA, Nvidia RAPIDS cugraph, 2020, https://github.com/rapidsai/cugraph.
[41] J. Zedlewski, End-to-end data science on GPUs with {RAPIDS}, 2020.
[42] A. Lancichinetti, S. Fortunato, F. Radicchi, Benchmark graphs for testing community detection algorithms, Phys. Rev. E 78 (4) (2008) 046110.
[43] T. Bodenheimer, M. Halappanavar, S. Jefferys, R. Gibson, S. Liu, P.J. Mucha, N. Stanley, J.S. Parker, S.R. Selitsky, FastPG: fast clustering of millions of single cells, 2020, Biorxiv.
[44] S. Bhowmick, S. Srinivasan, A template for parallelizing the louvain method for modularity maximization, in: Dynamics on and of Complex Networks, Vol. 2, Springer, 2013, pp. 111–124.
[45] M. Halappanavar, H. Lu, A. Kalyanaraman, A. Tumeo, Scalable static and dynamic community detection using grappolo, in: 2017 IEEE High Performance Extreme Computing Conference, HPEC, IEEE, 2017, pp. 1–6.
[46] M. Fazlali, E. Moradi, H.T. Malazi, Adaptive parallel louvain community detection on a multicore platform, Microprocess. Microsyst. 54 (2017) 26–34.
[47] J.J. Tithi, A. Stasiak, S. Aananthakrishnan, F. Petrini, Prune the unnecessary: Parallel pull-push louvain algorithms with automatic edge pruning, in: 49th International Conference on Parallel Processing, ICPP, 2020.

[48] C. Wickramaarachchi, M. Frincu, P. Small, V.K. Prasanna, Fast parallel algorithm for unfolding of communities in large graphs, in: High Performance Extreme Computing Conference, HPEC, 2014 IEEE, IEEE, 2014, pp. 1–6.
[49] X. Que, F. Checconi, F. Petrini, J.A. Gunnels, Scalable community detection with the louvain algorithm, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015, IEEE Computer Society, 2015, pp. 28–37.
[50] J. Zeng, H. Yu, Effectively unified optimization for large-scale graph community detection, in: 2019 IEEE International Conference on Big Data, Big Data, IEEE, 2019, pp. 475–482.
[51] S.-H. Bae, B. Howe, GossipMap: A distributed community detection algorithm for billion-edge directed graphs, in: Proceedings of The International Conference For High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 27.
[52] N. Buzun, A. Korshunov, V. Avanesov, I. Filonenko, I. Kozlov, D. Turdakov, H. Kim, Egolp: Fast and distributed community detection in billion-node social networks, in: Data Mining Workshop (ICDMW), 2014 IEEE International Conference On, IEEE, 2014, pp. 533–540.
[53] M. Ovelgönne, Distributed community detection in web-scale networks, in: Proceedings of The 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ACM, 2013, pp. 66–73.
[54] V. Prasanna, GoFFish: Graph-Oriented Framework for Foresight and Insight Using Scalable Heuristics, Tech. Rep., UNIVERSITY of SOUTHERN CALIFORNIA LOS ANGELES, 2015.
[55] X. Que, F. Checconi, F. Petrini, J.A. Gunnels, Scalable community detection with the louvain algorithm, in: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 2015, pp. 28–37.
[56] G.M. Slota, J.W. Berry, S.D. Hammond, S.L. Olivier, C.A. Phillips, S. Rajamanickam, Scalable generation of graphs for benchmarking HPC community-detection algorithms, in: Proceedings of The International Conference For High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–14.
[57] G. Karypis, K. Schloegel, V. Kumar, Parmetis: Parallel graph partitioning and sparse matrix ordering library, 1997, Version 1.0, Dept. of Computer Science, University of Minnesota.
[58] A. Abou-Rjeili, G. Karypis, Multilevel algorithms for partitioning power-law graphs, in: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2006, pp. 10–pp.
[59] J. Zeng, H. Yu, A scalable distributed louvain algorithm for large-scale graph community detection, in: 2018 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2018, pp. 268–278.
[60] C.Y. Cheong, H.P. Huynh, D. Lo, R.S.M. Goh, Hierarchical parallel algorithm for modularity-based community detection using GPUs, in: European Conference on Parallel Processing, Springer, 2013, pp. 775–787.
[61] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A.T. Riffel, et al., Gunrock: GPU graph analytics, ACM Trans. Parallel Comput. (TOPC) 4 (1) (2017) 1–49.
[62] Y. Pan, Multi-GPU Graph Processing (Ph.D. thesis), University of California, Davis, 2019.
[63] A. Bhowmick, S. Vadhiyar, Hysdetect: A hybrid CPU-gpu algorithm for community detection, in: 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics, HiPC, IEEE, 2019, pp. 2–11.

Nitin A. Gawande, Ph.D. is currently a Software Application Engineer with the Architecture and Workload Engineering Accelerated Computing Systems and Graphics group at Intel Corporation. Nitin's current research interests include design and implementation of scalable high performance computing algorithms on GPU accelerators. He has authored several peer reviewed journals and peer reviewed conference proceedings. He is a member of Association for Computing Machinery (ACM).

Sayan Ghosh is a Computer Scientist in the Data Sciences group at the Pacific Northwest National Laboratory (PNNL) in Richland, WA. His research interests are broadly in the application of parallel programming models for building scalable codes on supercomputers. He holds a Masters degree from University of Houston in Houston, TX and a Ph.D. degree (both in Computer Sciences) from Washington State University in Pullman, WA.

Mahantesh Halappanavar is a computer scientist and group leader of the Data Sciences and Machine Intelligence Group at the Pacific Northwest National Laboratory. He holds a joint appointment as an adjunct faculty in the School of Electrical Engineering and Computer Science at the Washington State University. His research focuses on developing efficient parallel graph algorithms and their applications to several domains including the analysis of electric power grids, sparse linear algebra, and cyber security.

Antonino Tumeo received M.S degree in Informatic Engineering, in 2005, and the Ph.D. degree in Computer Engineering, in 2009, from Politecnico di Milano in Italy. Since February 2011, he has been a research scientist in the PNNL's High Performance Computing group. He Joined PNNL in 2009 as a post doctoral research associate. Previously, he was a post doctoral researcher at Politecnico di Milano. His research interests are modeling and simulation of high performance architectures, hardware–software codesign, FPGA prototyping and GPGPU computing.

Ananth Kalyanaraman received the bachelor's degree from the Visvesvaraya National Institute of Technology, Nagpur, India, in 1998, and the MS and Ph.D. degrees from Iowa State University, Ames, in 2002 and 2006, respectively. Currently, he is an associate professor in the School of Electrical Engineering and Computer Science, Washington State University, Pullman and also holds a joint appointment at Pacific Northwest National Laboratory (PNNL). His research focuses on developing parallel algorithms and software for data-intensive problems originating in the areas of computational biology and graph-theoretic applications. He received the DOE Early Career Award, an Early Career Impact Award and two best paper awards. He serves on editorial boards of the IEEE Transactions on Parallel and Distributed Systems and the Journal of Parallel and Distributed Computing. He is a member of the AAAS, the ACM, the IEEE, the ISCB, and the SIAM.