2018 IEEE International Parallel and Distributed Processing Symposium

Distributed Louvain Algorithm for Graph Community Detection

Sayan Ghosh*, Mahantesh Halappanavar†, Antonino Tumeo†, Ananth Kalyanaraman*,
Hao Lu§, Daniel Chavarrià-Miranda‡, Arif Khan†, Assefaw H. Gebremedhin*

* Washington State University, Pullman, WA, USA {sghosh1, ananth, assefaw}@eecs.wsu.edu
† Pacific Northwest National Laboratory, Richland, WA, USA {hala, antonino.tumeo, ariful.khan}@pnnl.gov
‡ Trovares, Inc., Seattle, WA, USA daniel@trovares.com
§ Oak Ridge National Laboratory, Oak Ridge, TN, USA luh1@ornl.gov

***Abstract*—In most real-world networks, the 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 "related" to one another than they are to the rest of the network. The goodness of partitioning into communities is typically measured using a well known measure called *modularity*. However, modularity optimization is an NP-complete problem. In 2008, Blondel, et al. introduced a multi-phase, iterative heuristic for modularity optimization, 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.**

In this paper, we present the design of a distributed memory implementation of the Louvain algorithm for parallel community detection. Our approach begins with an arbitrarily partitioned distributed graph input, and employs several heuristics to speedup the computation of the different steps of the Louvain algorithm. We evaluate our implementation and its different variants using real-world networks from various application domains (including internet, biology, social networks). Our MPI+OpenMP implementation yields about 7x speedup (on 4K processes) for soc-friendster network (1.8B edges) over a state-of-the-art shared memory multicore implementation (on 64 threads), without compromising output quality. Furthermore, our distributed implementation was able to process a larger graph (uk-2007; 3.3B edges) in 32 seconds on 1K cores (64 nodes) of NERSC Cori, when the state-of-the-art shared memory implementation failed to run due to insufficient memory on a single Cori node containing 128 GB of memory.

***Index Terms*—Community detection, Parallel Louvain, Distributed graph clustering, Parallel graph heuristics.**

I. Introduction

Community detection is a widely used operation in graph analytics. Given a graph $G = (V, E)$, the goal of the community detection problem is to identify a partitioning of vertices into "communities" (or "clusters") such that related vertices are assigned to the same community and disparate/unrelated vertices are assigned to different communities. The community detection problem is different from the classical problem of graph partitioning in that neither the number of communities nor their size distribution is known *a priori*. Because of its ability to uncover structurally coherent modules of vertices, community detection has become a structure discovery tool in a number of scientific and industrial applications, including biological sciences, social networks, retail and financial networks, and literature mining. Comprehensive reviews on the various formulations, methods, and applications of community detection can be found in [9], [11], [23], [26].

Various measures have been proposed to evaluate the goodness of partitioning produced by a community detection method [16], [18], [21]. Of these measures, *modularity* is one that is widely used. Proposed by Newman [24], the measure provides a statistical way to quantify the goodness of a given community-wise partitioning, on the basis of the fraction of edges that lie within communities. Modularity has its limitations; more specifically, it suffers from a resolution limit [12]. Additionally, modularity optimization is an NP-complete problem [7]. Despite these limitations, the measure continues to be widely used in practice [11], [13]. Resolution-limit-free versions of modularity have been proposed [30]. Furthermore, numerous efficient heuristics 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. [5]. The method is a multi-phase, multi-iteration heuristic that starts from an initial state of $|V|$ communities (with one vertex per community) and iteratively improves the quality of community assignment until the gain in quality (i.e., modularity gain) becomes negligible. From a computation standpoint, this translates 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 [15], the Louvain method has been widely adopted by practitioners. Since its introduction to the field, there have been multiple attempts at parallelizing the Louvain heuristic (see Section II). To the best of our knowledge, the fastest shared memory multithreaded implementation of Louvain is the *Grappolo* software package [22]. The implementation was able to process a large real-world network (soc-friendster; 1.8B edges) in 812 seconds on a 20 core, 768 GB DDR3 memory Intel® Xeon™shared memory machine [14].

In this paper, we extend this line of work into the distributed memory domain. More specifically, we present a distributed memory implementation of the Louvain method for parallel community detection. One of the key challenges in the design

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00098

of an efficient distributed memory Louvain implementation is to enable efficient vertex neighborhood scans (for changes in neighboring community states), since with a distributed representation of the graph, communication overheads can become significant. Another key challenge is the frequency at which community states are accessed for queries and updates; the serial algorithm has the benefit of progressing from one iteration to the next in a synchronized manner (benefiting always from the latest of state information), while the cost of maintaining and propagating such latest information could become prohibitive in a distributed setting. The variable rates at which vertices are processed across the processor space presents another layer of challenge in the distributed setting. The approach proposed in this paper overcomes the above challenges using a combination of various heuristics.

Contributions: We make the following contributions in this paper:

- Discuss the design of distributed memory parallel Louvain algorithm (Section IV)
- Present effective heuristics for optimizing performance of distributed community detection (Section IV-B)
- Carry out performance analysis of the proposed algorithm using real-world moderate-large scale networks on 16 to 4K processes of NERSC Cori (Section V)

The remainder of the paper is organized as follows: After a brief review of related work in Section II, we provide in Section III essential preliminaries on the community detection problem along with a description of the serial Louvain heuristic and associated design challenges for parallelization. We present our distributed memory parallel algorithm and the heuristics we introduce to improve performance in Section IV. We provide an extensive performance evaluation on real-world networks in Section V. Section VI concludes the paper.

II. Related work

There have been a number of prior research on distributed parallel community detection [4], [8], [25], [27], [28], [31]. Among these, an MPI-based distributed memory Louvain implementation is reported in [28]. Like us, they split the vertices and their edge lists among the processes using a 1D decomposition. Therefore, our distribution strategies are similar. However, the overall methods are different, including the use of 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. The authors report the execution time of the uk-2007 real-world network (3.3B edges) to be ∼45 secs on 128 IBM® Power7™nodes. In comparison, we report an all inclusive execution time of ∼47 secs for uk-2007 on 128 processes using 8 Intel® Haswell nodes of NERSC Cori, and 4 OpenMP threads per process.

Another MPI implementation is discussed in [31], where ParMETIS [17] is used to near-optimally partition the graph among processes before the distributed memory community detection algorithm starts. Since graph partitioning is an NP-hard problem, we decided not to spend time finding a near-optimal graph partition, and work with a simpler distribution instead.

III. Preliminaries

A graph is represented by $G = (V, E)$, where V is the set of vertices and E is the set of edges. An edge between vertex i and j may have an associated edge weight $w_{i,j}$. The community detection problem is one of identifying a set of communities in an input graph, where the *communities* represent a partitioning of V. The goodness of clustering achieved by community detection can be measured by a global metric such as *modularity* [24]. More specifically, given a community-wise partitioning of an input graph, modularity measures the difference between the fraction of edges within communities compared to the expected fraction that would exist on a random graph with identical vertex and degree distribution characteristics. Given G and its adjacency matrix representation A, the modularity of G, denoted by Q, is given by:

$$Q = \frac{1}{2m}\sum_{i,j}(A_{ij} - \frac{k_i * k_j}{2m})\delta(c_i, c_j)$$
$$\text{where:}$$
$$m = \text{sum of all the edge-weights} \tag{1}$$
$$k_i = \text{weighted degree of vertex } i$$
$$c_i = \text{community that contains vertex } i$$
$$\delta(c_i, c_j) = 1 \text{ if } c_i = c_j, 0 \text{ otherwise.}$$

In practical terms, modularity depends on the sum of all edge weights between vertices within a particular community (denoted by e_{ij}), and sum of weights of all edges incident upon each community c (denoted by a_c). Viewed that way, Equation 1 can be written as Equation 2, where C denotes the set of communities.

$$Q = \sum_{c \in C}\left[\frac{e_{ij}}{2m} - \left(\frac{a_c}{2m}\right)^2\right]$$
$$\text{where:} \tag{2}$$
$$e_{ij} = \sum w_{ij} : \forall i, j \in c, \text{ and } \{i, j\} \in E$$
$$a_c = \sum_{i \in c} k_i$$

We use the formulation in Equation 2 in our implementations.

A. Serial Louvain algorithm

The Louvain method is iterative and consists of multiple phases. Each phase runs for a number of iterations until convergence. Initially, each vertex is assigned to a separate community. Within each iteration, all vertices are processed as follows: for a given vertex v, the gain in modularity (ΔQ) that would result in moving v to each of its neighboring communities is calculated; if the maximum of such gain is positive, then v is moved to that community from its current community. The phase is continued until the gain in modularity between any two successive iterations falls below a user-specified threshold (τ). When a phase ends, the graph for the next phase is rebuilt, by collapsing all vertices within a community into a single meta-vertex, and the process is continued until no appreciable gain in modularity is achieved between consecutive phases.

A pseudocode for the serial Louvain algorithm is shown in Algorithm 1.

Algorithm 1: Serial Louvain algorithm.
Input: Graph $G = (V, E)$, threshold τ
Input: Initial community assignment, C_{init}

```
1:  $Q_{prev} \leftarrow -\infty$
2:  $C_{prev} \leftarrow$ Initialize each vertex in its own community
3:  while true do
4:      for all $v \in V$ do
5:          $N(v) \leftarrow$ neighboring communities of $v$
6:          $targetComm \leftarrow \arg\max_{t \in N_v} \Delta Q(v$ moving to $t)$
7:          if the gain is positive then
8:              Move $v$ to $targetComm$ and update $C_{curr}$
9:      $Q_{curr} \leftarrow ComputeModularity(V, E, C_{curr})$
10:     if $Q_{curr} - Q_{prev} \leq \tau$ then
11:         break
12:     else
13:         $Q_{prev} \leftarrow Q_{curr}$
```

B. Challenges in distributed memory parallelization

The primary issue affecting the global modularity in distributed memory parallelization of the Louvain algorithm stems from concurrent community updates. A particular process only has the updated vertex-community association information from its last synchronization point. Between the last synchronization point and by the time the current process accesses a community, it is possible that a remote process has marked some updates for the community. However, these changes will be applied at the next synchronization point. Due to this lag of community update, the global modularity score (and overall convergence) of a distributed-memory parallel implementation of Louvain algorithm could be different from a similar serial or shared memory implementation. Lu et al. [22] discuss some challenges in parallelization such as negative gain and local maxima scenarios which are relevant for distributed memory cases as well.

There is significant communication overhead at every iteration of every phase, owing to exchange of community updates (vertices entering and leaving communities). Updated community information is required for calculating the cumulative edge weights within a community, and incident on a community, which are part of the modularity calculation. Therefore, at every iteration, we need updated community information of tail/ghost vertices (a vertex owned by another process, but is stored as part of the edge list in the current process). Also, if a locally owned vertex moves to another community that is owned by a remote process, then the degree and edge weights pertaining to that vertex need to be communicated to the target community owner as well. Modularity calculation also requires global accumulation of the weights, requiring collective communication operations. Finally, at the end of a phase, the graph is rebuilt, which entails communicating new vertex-community mappings to the respective owners of ghost vertices.

IV. Parallel Algorithm

In this section we discuss our parallel Louvain implementation and the various heuristics we introduce to optimize performance. We use p to denote the number of processes, and rank i to denote an arbitrary rank in the interval $[0, p-1]$.

Input Distribution: We distribute the input vertices and their edge lists evenly across available processes, such that each process receives roughly the same number of edges; no clever graph partitioning is performed. Each process stores a subset of vertices that it owns, and 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. Similarly, each process owns an arbitrary subset of communities (set initially to equal number of communities per process), and also keeps track of 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.

A. Overview of the parallel algorithm

As mentioned earlier, the Louvain algorithm comprises multiple phases, and each phase is run for a number of iterations. Initially, each vertex is in its own community, and as community detection progresses, vertices migrate by entering and leaving communities. Each vertex resides in one community at the start of an iteration, and decides on which of its neighboring communities to move to by the end of an iteration. Algorithm 2 shows a high-level description of the parallel Louvain algorithm executing on a process. In this pseudocode, each iteration of the **while** loop corresponds to a Louvain "phase".

Algorithm 2: Parallel Louvain Algorithm (at rank i).
Input: Local portion $G_i = (V_i, E_i)$ of the graph $G = (V, E)$
Input: Threshold, τ (default: 10^{-6})

```
1:  $C_{curr} \leftarrow \{\{u\} | \forall u \in V\}$
2:  $\{currMod, prevMod\} \leftarrow 0$
3:  while true do
4:      $currMod \leftarrow LouvainIteration(G_i, C_{curr})$
5:      if $currMod - prevMod \leq \tau$ then
6:          break and output the final set of communities
7:      $BuildNextPhaseGraph(G_i, C_{curr})$
8:      $prevMod \leftarrow currMod$
```

Algorithm 2 shows the two major steps of the parallel Louvain algorithm. The first step involves invoking the Louvain iteration, which runs the Louvain heuristic for modularity maximization. The second step is graph reconstruction, where vertices in each cluster are collapsed into a single meta-vertex, compacting the graph. In what follows, we describe these two steps more in detail.

a) Louvain iteration: Algorithm 3 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 (a_c) is stored locally (as part of the vector C_{info} in Algorithm 3). 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 (owing to graph compaction), we perform a single (one-time per phase) send-receive communication step to exchange these ghost coordinate information (Algorithm 4). 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 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 main body of each Louvain iteration consists of the following major steps (see Algorithm 3):

i) At the beginning of each iteration, information about ghost vertices (i.e., their latest community assignments) are received at each process (lines 4-5);
ii) Using the latest vertex information, compute the new community assignments for all local vertices (lines 7-9). This is a local computation step;
iii) Send all updated information for ghost communities to their owner processes, and receive and updated information on any local communities that were updated remotely (lines 10-11);
iv) Compute the global modularity based on the new community state (lines 12-13); and
v) If the net modularity gain (ΔQ) achieved relative to the previous iteration is below the desired threshold τ, then terminate the phase (continue otherwise).

b) Graph reconstruction: The communities found at the end of the current phase are considered as new vertices for the compressed graph for the next phase. Edges within a community form a self loop around the community, whereas the weights of edges between communities are added and a single edge is placed between them with the cumulative weight. The graph reconstruction phase is shown via an example in Fig. 1. 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.

Algorithm 3: Algorithm for the Louvain iterations of a phase at rank i.
Output: Modularity at the end of the phase.

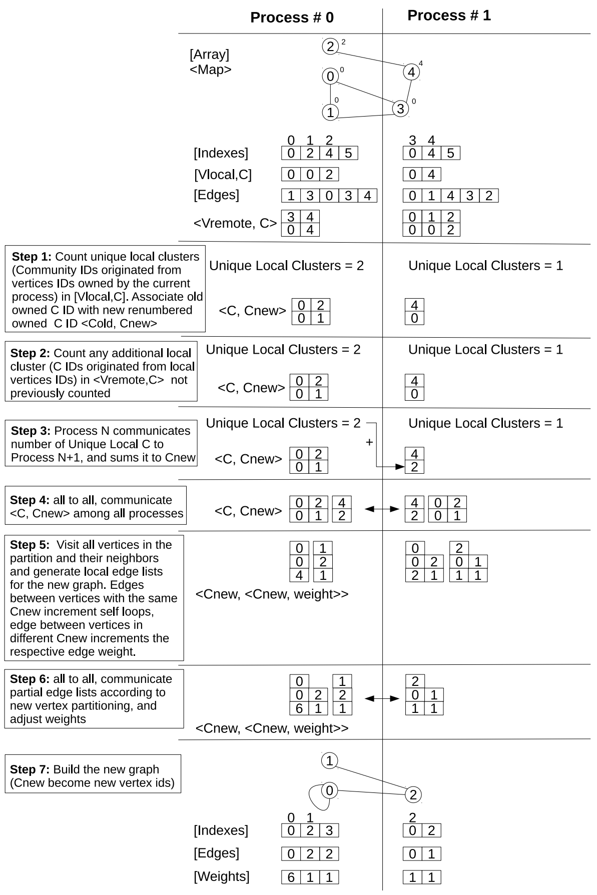
```
1:  function LouvainIteration($G_i, C_{curr}$)
2:  $V_g \leftarrow ExchangeGhostVertices(G_i)$
3:  while true do
4:      send latest information on those local vertices that are
        stored as ghost vertices on remote processes
5:      receive latest information on all ghost vertices
6:      for $v \in V_i$ do
7:          Compute $\Delta Q$ that can be achieved by moving $v$ to each
            of its neighboring communities
8:          Determine target community for $v$ based on the migration
            that maximizes $\Delta Q$
9:          Update community information for both the source and
            destination communities of $v$
10:     send updated information on ghost communities to owner
        processes
11:     $C_{info} \leftarrow$ receive and update information on local
        communities
12:     $currMod_i \leftarrow$ Compute modularity based on $G_i$ and $C_{info}$
13:     $currMod \leftarrow$ all-reduce: $\sum_{\forall i} currMod_i$
14:     if $currMod - prevMod \leq \tau$ then
15:         break
16:     $prevMod \leftarrow currMod$
17: return $prevMod$
```

Algorithm 4: Algorithm to receive information about ghost vertices from remote (owner) processes.
Input: Local portion $G_i(V_i, E_i)$ (in CSR format)
Output: List V_g of ghost vertices

```
1:  function $ExchangeGhostVertices(G_i)$
2:  for $v \in V_i$ do
3:      $[e0, e1] \leftarrow getEdgeRangeForVertex(v)$
4:      for $u \in [e0, e1]$ do
5:          $owner \leftarrow G_i.getOwner(u)$
6:          if $owner \neq me$ then
7:              $vmap[owner] \leftarrow vmap[owner] \cup \{u\}$
8:  for $j \in [0, p-1]$ do
9:      if $j \neq me$ then
10:         send $vmap[j]$ to rank $j$
11:         receive data in $V_g[j]$ list
12: return $V_g$
```

The distributed graph reconstruction process proceeds according to the following steps, as corresponded by Fig. 1.

1) Each process counts its unique local clusters, which are renumbered starting from 0. Renumbering is performed with a map that associates the old community ID with the new ID.
2) 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.
3) Local unique clusters are renumbered globally: this is achieved using a parallel prefix sum computation on the number of unique clusters.



Process # 0
Process # 1
[Array]
<Map>
[Indexes]
[Vlocal,C]
[Edges]
<Vremote, C>
Step 1: Count unique local clusters (Community IDs originated from vertices IDs owned by the current process) in [Vlocal,C]. Associate old owned C ID with new renumbered owned C ID <Cold, Cnew>
Unique Local Clusters = 2
Unique Local Clusters = 1
<C, Cnew>
Step 2: Count any additional local cluster (C IDs originated from local vertices IDs) in <Vremote,C> not previously counted
Step 3: Process N communicates number of Unique Local C to Process N+1, and sums it to Cnew
Step 4: all to all, communicate <C, Cnew> among all processes
Step 5: Visit all vertices in the partition and their neighbors and generate local edge lists for the new graph. Edges between vertices with the same Cnew increment self loops, edge between vertices in different Cnew increments the respective edge weight.
<Cnew, <Cnew, weight>>
Step 6: all to all, communicate partial edge lists according to new vertex partitioning, and adjust weights
Step 7: Build the new graph (Cnew become new vertex ids)
[Indexes]
[Edges]
[Weights]

Fig. 1. Graph 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.

4) Processes are involved in communicating the new global community IDs for the local partition. Only the new community IDs that replaces the old community IDs used in other processes need to be communicated.
5) 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.
6) 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).
7) New arrays for indices and vertices of the coarsened graph can thus finally be rebuilt from the edge lists.

B. Heuristics for performance optimization

We present two heuristics which further improve the overall execution times of the distributed Louvain algorithm by reducing the number of iterations within a phase.

a) Threshold Cycling: The Louvain algorithm uses a threshold τ to decide termination—more specifically, if the net modularity gain achieved between any two successive phases (Algorithm 2) falls below τ then the algorithm is terminated (achieves convergence). (Note that the same threshold is also used between consecutive iterations of a phase in Algorithm 3 to terminate a phase.) Typically, this τ parameter is kept fixed throughout the execution.

We extend the concept presented by Lu et al. [22] for a multithreaded Louvain algorithm implementation, in tuning the threshold across phases. The main idea is that during the initial phases, when the graph is relatively large, the threshold is also kept large, and is reduced incrementally for the later phases. The point here is that if the threshold is small, then the Louvain algorithm per phase will typically undergo more iterations before it can exit; however, a higher threshold could translate to lesser number of iterations to convergence. Such savings in the number of iterations are likely to result in larger performance savings in the earlier phases when the graph is still large.

We implemented a scheme in which the threshold is modulated in a cyclical fashion across phases. A range of threshold values are invoked in successive phases after every N phases, where N is predetermined. This is demonstrated in Fig. 2, where phases 0–2, 3–6, 7–9 and 10–12 have respective thresholds of 10^{-3}, 10^{-4}, 10^{-5} and 10^{-6}. This pattern is again repeated from phase 13 and so on.

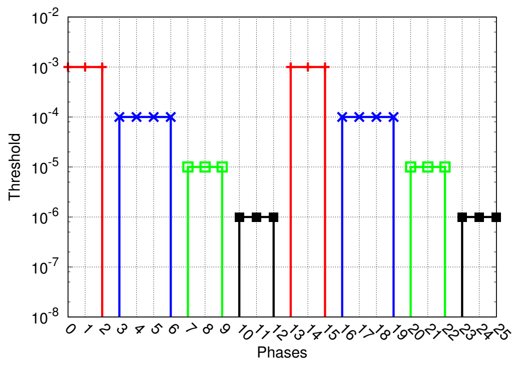


Fig. 2. Threshold cycling illustration.

b) Early Termination: In our parallel Louvain algorithm, one of the major contributors to communication cost is expected to result from communicating ghost vertex information across processes. This cost can easily become a bottleneck if the original partitioning of the input graph has a large fraction of edges between vertices that reside in different processes. However, after experimenting with our multithreaded Louvain algorithm [22] with numerous inputs, we made a critical observation: that, the rate at which the overall modularity increases significantly slows down as the number of iterations increases within a phase. This *diminishing returns* property in quality is captured in almost all of the modularity evolution charts presented in [22]. This happens because within a phase, the rate at which vertices change their community affiliation tends to drastically reduce as the iterations progress within a phase. In other words, vertices tend to hop around initially but soon collocate with their community partners, thereby becoming less likely to move in the later stages.

We present a heuristic to take advantage of the above observation. We devise a probabilistic scheme by which a vertex decides to stay "active" or become "inactive", at any given iteration. Being "active" implies that the vertex will participate in the computation within the main body of the Louvain iteration (Algorithm 3; lines 7-10), and will recompute its current community affiliation. Alternatively, if the vertex is "inactive", it will be dropped from the processing queue during that iteration. Note that by making a vertex inactive during an iteration, we can save on all the potential computation *and* communication that it generates. The savings can be particularly significant for larger degree vertices. To identify which vertices to make inactive, we take advantage of the above observation by looking at the most recent activity of that vertex; intuitively, if the vertex has not moved lately then we reduce the probability that it will stay active.

More specifically, consider a vertex v. Let $C_{v,j}$ denote the community containing v at the end of any given iteration j. Let the probability that v is *active* during iteration k be denoted by $P_{v,k}$. We define $P_{v,k}$ as follows:

$$P_{v,k} = \begin{cases} P_{v,k-1} * (1-\alpha), & \text{if } C_{v,k-1} = C_{v,k-2} \\ 1, & \text{otherwise} \end{cases} \tag{3}$$

where α is a real number between 0 and 1. The idea is to rapidly decay the probability as a vertex continues to stay in its current community. As α approaches zero, it becomes similar to the baseline scheme; and as it approaches one, it becomes highly aggressive in terminating vertices early on during execution, with the potential risk of compromising on quality. Consequently, we call this probabilistic heuristic the "early termination" (ET) heuristic.

To study the effect of the α parameter on the performance-quality trade-off, we modified our multithreaded implementation of [22] with this heuristic and tested it on two real-world inputs. The results are tabulated in Table I. As can be observed, the heuristic demonstrates the ability to achieve significant run-time savings with negligible loss in quality. The savings vary by the input—for CNR ($3.2M$ edges), we observe about 2x speedup in run-time (from $\alpha = 0$ to 1), while for Channel ($42.7M$ edges), we see a 58.27x speedup. CNR has small world characteristics, while Channel has a banded structure.

Consequently, we incorporate this heuristic into our distributed implementation. More specifically, we changed the main computation `for` loop of Algorithm 3, so that each vertex marks itself first as active or inactive based on the probabilistic scheme, and subsequently includes itself for further processing or not. We developed two minor variants to the above ET idea (labeled, ET and ETC, respectively):

First, when the probability for a given vertex becomes less than 2%, we label it inactive.

Secondly, the early termination scheme can also be combined with another option to provide reasonably better performance. We provide an option to calculate the percentage of global inactive vertices, and if 90% of the vertices are inactive in a particular phase, then the program exits. This option requires an extra remote communication, involving global summation of inactive vertices. In certain cases, we observe early termination with remote communication to be around ∼1.25-2.3x better than using early termination alone.

Further sophistication in the implementation is possible. Note that if α is 1, then once a vertex marks itself as inactive, it will stay inactive for ever. In fact, we should be able to safely argue that this property will hold (with high probability) for large α values as it nears 1. Secondly, any communication that relates to inactive vertices can be prevented/preempted by communicating the ghost vertex IDs that have become inactive to other processes that still think they need them.

Our distributed memory Louvain algorithm code is available for download under the BSD 3-clause license from: http://hpc.pnl.gov/people/hala/grappolo.html.

V. Experimental evaluation

We used real-world graphs for our experimental evaluations (see Table II). As far as we know, currently no MPI-based distributed memory Louvain algorithm implementation is publicly available. Therefore, to compare our results, we use the multithreaded implementation of the Louvain algorithm, namely Grappolo [22]. In our evaluations, we include both performance (execution time) and output quality (modularity), and also report on the number of iterations used by respective implementations. Below we summarize the descriptors/legends used in the figures/tables in this section to refer to the different variants of our parallel algorithm (discused in Section IV-B).

- `Baseline`: the main parallel version (Algorithm 2) without the heuristics in Section IV-B.
- `Threshold Cycling`: version with threshold cycling enabled.
- `ET`: version with adaptive early termination, which requires an input parameter (α). We report ET performance with $\alpha = 0.25$ and $\alpha = 0.75$.
- `ETC`: variant of ET with an extra communication step to gather inactive vertex count. We report ETC performance with $\alpha = 0.25$ and $\alpha = 0.75$.

Experimental setup: We used the NERSC Cori supercomputer for our distributed/shared memory evaluations. NERSC Cori is a 2,388-node Cray® XC40™machine with dual-socket Intel® Xeon™E5-2698v3 (Haswell) CPUs at 2.3 GHz per node, 32 cores per node, 128 GB main memory per node, 40 MB L3 cache/socket and the Cray® XC™series interconnect (Cray® Aries™with Dragonfly topology). We use cray-mpich/7.6.0 as our MPI implementation, and Intel® 17.0.2 compiler with `-O3 -xHost` compilation option to build the codes.

Our primary testset consists of 12 graphs collected in their native formats from four sources: UFL sparse matrix collection [10], Network repository [29], SNAP [20] and LAW [6]. The graphs are listed in Table II, along with their respective output modularity as reported by Grappolo (using 1 thread).

TABLE I
PRELIMINARY EVALUATION OF OUR ADAPTIVE EARLY TERMINATION HEURISTIC USING OUR MULTITHREADED IMPLEMENTATION. ALL RUNS WERE EXECUTED ON 8 CORES OF AN INTEL® XEON™ MACHINE. THE INPUTS, CNR AND CHANNEL, ARE BOTH NETWORKS DOWNLOADED FROM [10], WITH $325K$ AND $4.8M$ VERTICES RESPECTIVELY.

	Input: CNR			Input: Channel		
α	Modularity	Time (in sec.)	No. iterations	Modularity	Time (in sec.)	No. iterations
1.0	0.91265	2.25	20	0.94314	1.73	28
0.9	0.91282	2.48	74	0.9432	6.69	76
0.8	0.91277	2.64	66	0.94329	6.71	72
0.7	0.91278	2.60	66	0.94349	8.44	88
0.6	0.91279	2.71	64	0.94334	11.58	117
0.5	0.91278	2.73	61	0.94333	15.56	152
0.4	0.91285	2.69	60	0.94352	20.48	195
0.3	0.91280	2.80	58	0.94314	25.61	232
0.2	0.91284	2.90	59	0.94312	39.44	348
0.1	0.91286	3.42	57	0.94338	69.26	583
0.0	0.91286	5.42	63	0.94369	100.82	232

TABLE II
TEST GRAPHS, LISTED IN ASCENDING ORDER OF EDGES.

Graphs	#Vertices	#Edges	Modularity
channel	4.8M	42.7M	0.943
com-orkut	3M	117.1M	0.472
soc-sinaweibo	58.6M	261.3M	0.482
twitter-2010	21.2M	265M	0.478
nlpkkt240	27.9M	401.2M	0.939
web-wiki-en-2013	27.1M	601M	0.671
arabic-2005	22.7M	640M	0.989
webbase-2001	118M	1B	0.983
web-cc12-PayLevelDomain	42.8M	1.2B	0.687
soc-friendster	65.6M	1.8B	0.624
sk-2005	50.6M	1.9B	0.971
uk-2007	105.8M	3.3B	0.972

We converted the test graphs from their various native formats to an edge list based binary format, and used the binary file as an input in our implementation. Since we make use of MPI I/O for reading the input file in parallel (and follow best practices), our overall I/O time is about 1-2% of the overall execution time. Since our implementation also uses OpenMP for multithreading, we set either 2 or 4 threads per process, in order to use all 64 threads per node of Cori.

Comparison on a single node: To assess the overhead of our MPI+OpenMP distributed memory Louvain implementation, we compared it with the multithreaded implementation from the Grappolo software package, on a single Cori node, using a single process and multiple threads. Table III shows the runtimes in seconds of our distributed memory implementation and the shared memory (Grappolo) implementation for the input graph soc-friendster (1.8B edges). The table shows that performance of the pure OpenMP version is about 2.3x better than our distributed version on all 32 cores of the node. On the other hand, the distributed version shows better scaling with the number of threads (about 4x speedup on 64 threads relative to 4 threads, whereas the shared-memory version scales to about 2x). In all these runs, the modularity difference was found to be under 1%. Furthermore, the distributed version obtains a speedup of up to 7x compared to the optimized shared-memory version on 64 threads, when we scale out to 4K processes on 256 nodes (see Fig. 3).

TABLE III
DISTRIBUTED MEMORY VS SHARED MEMORY (GRAPPOLO) PERFORMANCE (RUNTIME) OF LOUVAIN ALGORITHM ON A SINGLE CORI NODE USING 4-64 THREADS. THE INPUT GRAPH IS SOC-FRIENDSTER (1.8B EDGES).

#Threads	Distributed memory (sec.)	Shared memory (sec.)
4	6,082.25	1,216.54
8	3,615.52	843.37
16	2,252.09	725.26
32	1,515.24	689.38
64	1,303.98	554.52

A. Strong scaling

We report the total runtime (inclusive of the time to read the input graph, perform modularity maximization and graph reconstruction) for our test graphs in Fig. 3. We observe that the process end points of best speedup vary by the input, with moderate/large inputs showing reasonable scalability up to 1K-2K processes. However, some graphs such as sk-2005, have relatively low number of iterations per phase, which indicates that there is not enough work to utilize the increased parallelism beyond a certain point. These end points in scaling are to do with the balance between computation and communication times. For instance, we used HPCToolkit [1] to profile the Baseline version on soc-friendster on 256 processes (32 nodes). Analysis shows that 98% of the entire execution time is spent in the main body of the Louvain iterations (with 1% in graph rebuilding and another 1% for reading the input graph using MPI I/O routines). Of the 98%, roughly 34% is used in communicating community related information, and 40% is spent in the reduction operation (line#13 of Algorithm 3); whereas 22% of the time is used in computation (lines #6-#9).

To compare the different versions, we calculated speedup as the ratio between the Baseline execution time on 16-128 processes and the execution time for the fastest running version observed for a particular input. Table IV shows these results. It can be seen that early termination versions (ET or ETC) deliver the best performance in most cases.

B. Weak scaling

For weak scaling analysis, we use GTgraph synthetic graph generator suite [3] to generate graphs according to DARPA HPCS SSCA#2 benchmark [2]. Graphs following SSCA#2

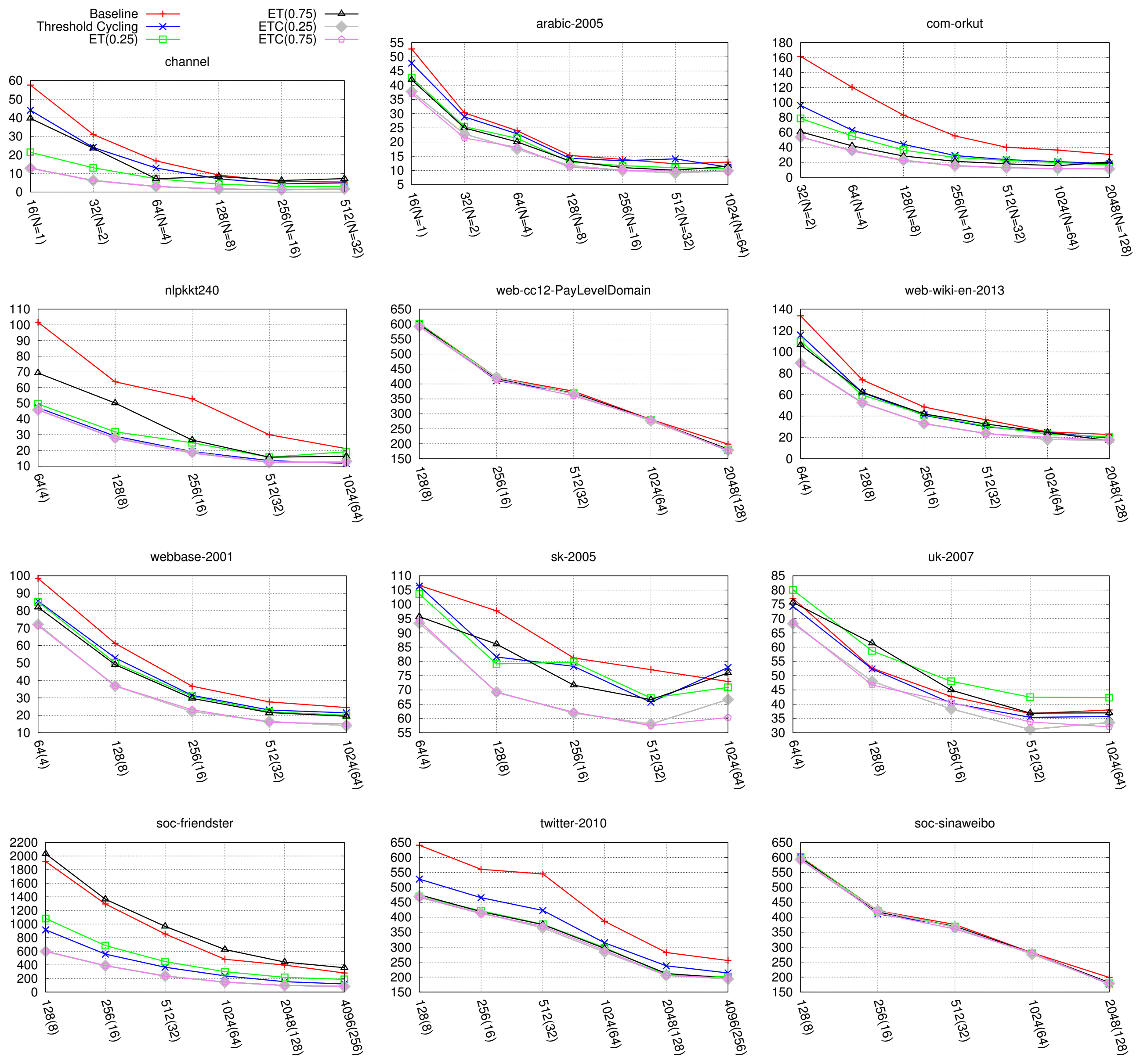


Fig. 3. Execution times of our distributed Louvain implementation for graphs listed in Table II. **X-axis**: Number of processes (and nodes), **Y-axis**: Execution time (in secs.).

benchmark are comprised of random-sized cliques, with various parameters to control the amount of vertex connections and inter-clique edges, along with other options to set the maximum sizes of clusters and cliques.

We fix the maximum clique size of the generated graphs (of various dimensions) to 100 and deliberately keep inter-clique edge probability low to enforce good community structure. Each graph is executed on a different combination of processes (and nodes), such that overall work-per-process is fixed. A list of the generated graphs along with the process-node configuration on which they are run is provided in Table V. Our distributed implementation reported exact same convergence criteria for each graph listed in Table V, since the underlying structures of the graphs are similar, despite the difference in sizes. The weak scaling results we obtained are summarized in Fig. 4. The figure shows nearly constant execution time for the Baseline versions of our distributed Louvain implementation using input SSCA#2 graphs of varying sizes and varying process counts (1-512).

C. Analysis

a) Benefits of Threshold Cycling: The Threshold Cycling scheme provided significant performance (runtime) benefit (compared to Baseline) with less than 3% decrease in modu-

TABLE IV
VERSIONS YIELDING THE BEST PERFORMANCE OVER THE BASELINE VERSION (RUN ON 16-128 PROCESSES) FOR INPUT GRAPHS (LISTED IN ASCENDING ORDER OF EDGES).

Graphs	Best speedup	Version
channel	46.18x	ETC(0.25)
com-orkut	14.6x	ETC(0.75)
soc-sinaweibo	3.4x	Threshold Cycling
twitter-2010	3.3x	ETC(0.25)
nlpkkt240	8.68x	Threshold Cycling
web-wiki-en-2013	7.92x	ET(0.75)
arabic-2005	5.8x	ETC(0.25)
webbase-2001	7x	ETC(0.25)
web-cc12-PayLevelDomain	3.75x	ETC(0.25)
soc-friendster	23x	ETC(0.25)
sk-2005	1.8x	ETC(0.75)
uk-2007	2.4x	ETC(0.75)

TABLE V
GTGRAPH SSCA#2 GENERATED GRAPH DIMENSIONS AND ASSOCIATED INFORMATION.

Name	#Vertices	#Edges	Modularity	#Processes (Nodes)
Graph#1	5M	333.7M	0.999981	1(1)
Graph#2	10M	660.7M	0.999990	32(2)
Graph#3	50M	3.3B	0.999998	208(13)
Graph#4	100M	6.6B	0.999999	448(28)
Graph#5	150M	6.9B	0.999999	512(32)

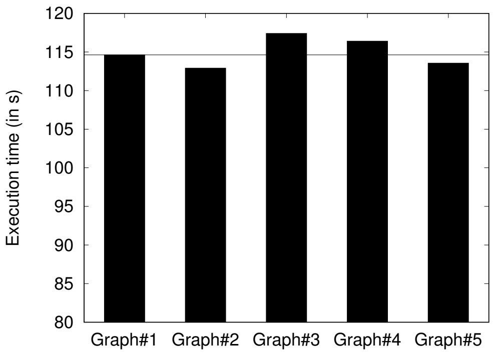


Fig. 4. Weak scaling of distributed Louvain implementation (Baseline) on GTgraph generated SSCA#2 graphs. **X-axis**: Input graphs listed in Table V, **Y-axis**: Execution time (in secs.).

larity for over 90% of the test graphs. Meanwhile, threshold cycling performed only marginally better for the soc-sinaweibo and web-wiki-en-2013 graphs. These graphs ran for only 3 or 4 phases, and the Louvain algorithm converged before ending a cycle of threshold modifications. In such cases, our distributed implementation always forces Louvain iteration to run once more with the lowest threshold (default $\tau = 10^{-6}$), to ensure acceptable modularity. Hence, in such cases, threshold cycling yields only nominal benefit.

b) Benefits of Early Termination: The runtime charts in Fig. 3 show that the early termination versions (ET or ETC) provide the best performance for most input graphs. We discuss the modularity growth and iterations per phase characteristics on 64 processes for two of the test graphs—nlpkkt240 and web-cc12-PayLevelDomain.

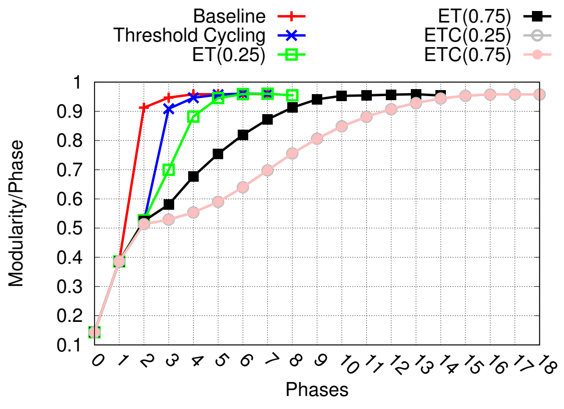
These results are shown in Figs. 5a and 5b (for nlpkkt240) and Figs. 6a and 6b (for web-cc12-PayLevelDomain). Generally, we observed one of two trends among all the test graphs: one in which ET with $\alpha = 0.25$ performs better than ET with $\alpha = 0.75$ (Figures 5a and 5b), and another in which the converse happens—(Figs. 6a and 6b).

In Fig. 5a we observe slow growth in modularity of ET(0.75) across many more phases (which increases the overall execution time) compared to ET(0.25). Also, we observe significantly more iterations per phase for ET(0.75) compared to ET(0.25) in Fig. 5b. It is to be noted that although the total number of phases of ET(0.75) is 2.6x, and total iterations is 1.3x to that of Baseline, the overall execution time of ET(0.75) is still about 1.47x better than Baseline. This is due to the prevalence of a number of inactive vertices per phase, the time spent per phase is significantly less than the Baseline version. With α close to 1, the scheme of labeling vertices as inactive becomes more aggressive. This hurts the convergence characteristics as evidenced by a higher number of phases for ET(0.75), as there are fewer (active) vertices at every phase which can move to other communities, maximizing ΔQ.

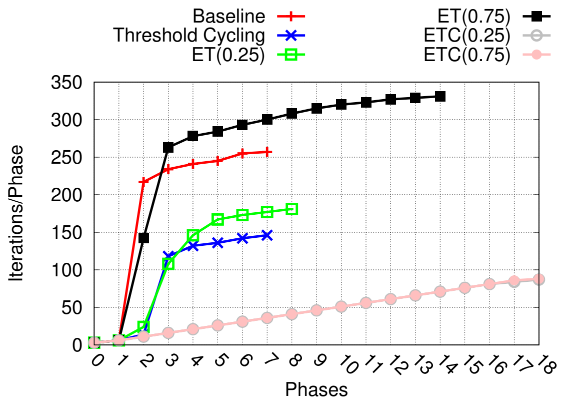
However, we observe an interesting phenomenon with ET, when the optional communication step is performed (i.e., version ETC). In Fig. 5b, we see that ETC(0.25) and ETC(0.75) display very similar performances, whereas ET(0.25) and ET(0.75) were quite different. The remote communication step computes the global number of vertices that are inactive, and if it is more than 90%, then Louvain iteration exits. This is quite different from the ET counterparts, as ET still relies comparing consecutive modularities to a fixed τ per iteration in a phase. Fig. 5b shows a linear increase in modularity/iterations, due to this exit condition, and yields about 20-30% performance benefit as compared to using ET alone.

For web-cc12-PayLevelDomain, we observe from Figures 6a and 6b that the aggressive version of ET with $\alpha = 0.75$ (denoted as ET(0.75)) performs better than ET(0.25), at the expense of 4% decrease in modularity. The performance of ET(0.75) is 16% better than ET(0.25) owing to lesser number of iterations per phase.

We also compared our distributed memory ET version directly against the shared memory results of Table I. For the CNR input, we varied α from 0 (least aggressive) to 1 (most aggressive). The results showed a reduction in runtime from 0.523 seconds down to 0.488 seconds ($\sim$6.7% improvement), largely owing to the reduction in the number of iterations (from 37 down to 24); while the modularity values were largely consistent up to the second decimal place. The time improvement is more modest compared to the shared memory results of Table I. Note that the shared memory version deploys a different set of heuristics such as coloring and vertex following [22], and therefore the behavior is expected to be different. Combining ET with Threshold Cycling yields better performance in some cases, as shown in Table VI. However, in nearly all ETC cases, we did not see any benefit of adding Threshold Cycling, because the exit criteria of ETC is not τ-based.

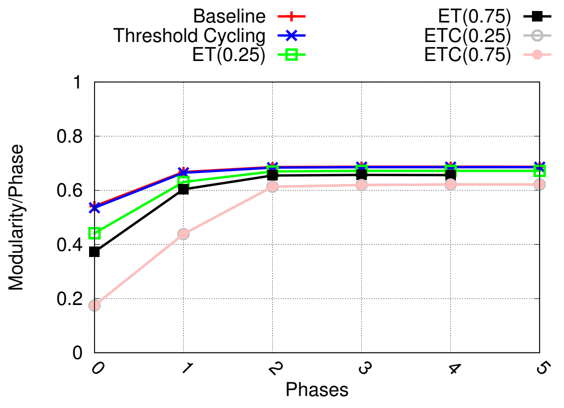


(a) Modularity growth.

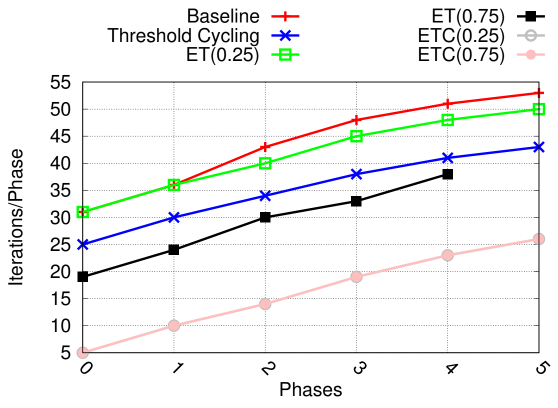


(b) Number of iterations.

Fig. 5. Convergence characteristics of nlpkkt240 (401.2M edges) on 64 processes.



(a) Modularity growth.



(b) Number of iterations.

Fig. 6. Convergence characteristics of web-cc12-PayLevelDomain (1.2B edges) on 64 processes.

TABLE VI
PERFORMANCE OF ET(0.25) COMBINED WITH THRESHOLD CYCLING FOR SOC-FRIENDSTER (1.8B EDGES). RELATIVE PERCENTAGE GAINS IN PERFORMANCE ARE IN BRACES.

Processes (Nodes)	Execution time ET(0.25) (secs.)	Execution time ET(0.25) + Threshold Cycling (secs.)
256(16)	683.996	614.788 (10%)
512(32)	448.048	398.773 (11%)
1024(64)	299.744	264.645 (12%)
2048(128)	216.606	195.127 (10%)
4096(256)	186.869	167.851 (10%)

TABLE VII
QUALITY COMPARISONS WITH GROUND TRUTH COMMUNITY INFORMATION.

#Vertices	#Edges	Precision	F-score
350K	34.73M	0.980889	0.990352
600K	58.91M	0.981865	0.990849
1M	98.12M	0.962938	0.981119
1.5M	147.13M	0.937488	0.967736
2M	196.45M	0.896050	0.945176

D. *Quality assessment*

In order to assess the quality of our distributed Louvain implementation, we compare our results against known ground truth communities for a variety of networks generated by the LFR benchmark [19]. When the quality assessment feature is turned on, our implementation performs extra collective operations per Louvain method phase to gather the vertex-community associations of the current graph into the root process.

We follow the exact same methodology to calculate F-score as discussed in [14], using the statistical measures of precision and recall, which are computed from the community assignment comparisons of the ground truth data with our distributed implementation. For the current set of LFR benchmark networks, we ran our distributed implementation on 2 nodes with 16 processes per node (with 2 OpenMP threads per process). As shown in Table VII, high F-score and precision (recall was found to be 1.0 for every case) corresponds to high quality solution as compared to ground truth community assignments. We also observed nearly identical F-score results reported by Grappolo for the same LFR benchmark networks.

VI. Conclusion

We presented a distributed memory implementation of the Louvain algorithm for parallel graph community detection. We introduced several heuristics, the inclusion of which was found crucial for improving performance and scalability. Our parallel implementation demonstrated speedups of 1.8x-46.18x (using up to 4K processes) relative to the Baseline version, for a wide variety of real-world networks. Modularities obtained by the different versions of our parallel algorithm are in most cases comparable to the best modularities obtained by a state-of-the-art multithreaded Louvain implementation. We believe the detailed discussion of the parallel implementation, the heuristics introduced, and the experimental analysis provided in this paper will benefit a wider range of graph algorithms which also have a greedy iterative structure with vertex-centric computations.

We plan to extend this work in several ways. One direction we are investigating is incorporation of other heuristics, including the use of distance-1 coloring to ensure that the set of vertices that are proceeed in parallel for community assignments are mutually non-adjacent and hence independent. This may lead to faster convergence. Apart from this, in order to make our implementation more scalable, we are considering neighborhood collective operations introduced in MPI-3.

Acknowledgment

We used resources of the NERSC facility, supported by U.S. DOE SC under Contract No. DE-AC02-05CH11231. This work was partially supported by NSF CAREER award IIS-1553528; DOE award DE-SC-0006516; ECP (17-SC-20-SC), a collaborative effort of U.S. DOE SC and NNSA at DOE PNNL.

References

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

[2] David A Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *International Conference on High-Performance Computing*, pages 465–476. Springer, 2005.

[3] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[4] Seung-Hee Bae and Bill 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*, page 27. ACM, 2015.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

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

[7] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[8] Nazar Buzun, Anton Korshunov, Valeriy Avanesov, Ilya Filonenko, Ilya Kozlov, Denis Turdakov, and Hangkyu Kim. Egolp: Fast and distributed community detection in billion-node social networks. In *Data Mining Workshop (ICDMW), 2014 IEEE International Conference on*, pages 533–540. IEEE, 2014.

[9] Michele Coscia, Fosca Giannotti, and Dino Pedreschi. A classification for community discovery methods in complex networks. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 4(5):512–546, 2011.

[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

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

[12] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[13] Benjamin H Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Physical Review E*, 81(4):046106, 2010.

[14] Mahantesh Halappanavar, Hao Lu, Ananth Kalyanaraman, and Antonino Tumeo. Scalable static and dynamic community detection using grappolo. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–6. IEEE, 2017.

[15] Darko Hric, Richard K Darst, and Santo Fortunato. Community detection in networks: Structural communities versus ground truth. *Physical Review E*, 90(6):062805, 2014.

[16] Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical Review E*, 83(1):016107, 2011.

[17] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[18] David Krackhardt and Robert N Stern. Informal networks and organizational crises: An experimental simulation. *Social psychology quarterly*, pages 123–140, 1988.

[19] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008.

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

[21] Zhenping Li, Shihua Zhang, Rui-Sheng Wang, Xiang-Sun Zhang, and Luonan Chen. Quantitative function for community detection. *Physical review E*, 77(3):036109, 2008.

[22] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[23] Mark EJ Newman. Communities, modules and large-scale structure in networks. *Nature physics*, 8(1):25, 2012.

[24] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[25] Michael 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*, pages 66–73. ACM, 2013.

[26] Mason A Porter, Jukka-Pekka Onnela, and Peter J Mucha. Communities in networks. *Notices of the AMS*, 56(9):1082–1097, 2009.

[27] Viktor Prasanna. Goffish: Graph-oriented framework for foresight and insight using scalable heuristics. Technical report, UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES, 2015.

[28] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. Scalable community detection with the louvain algorithm. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 28–37. IEEE, 2015.

[29] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[30] Vincent A Traag, Paul Van Dooren, and Yurii Nesterov. Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1):016114, 2011.

[31] Charith Wickramaarachchi, Marc Frincu, Patrick Small, and Viktor K Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.