2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Parallel Heuristics for Scalable Community Detection

Hao Lu, Ananth Kalyanaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA
Email: {luhowardmark,ananth}@wsu.edu

Mahantesh Halappanavar, Sutanay Choudhury
Computational Sciences and Mathematics Division
Pacific Northwest National Laboratory
Richland, WA
Email: {hala,sutanay.choudhury}@pnnl.gov

***Abstract*—Community detection has become a fundamental operation in numerous graph-theoretic applications. It is used to reveal natural divisions that exist within real world networks without imposing prior size or cardinality constraints on the set of communities. Despite its potential for application, there is only limited support for community detection on large-scale parallel computers, largely owing to the irregular and inherently sequential nature of the underlying heuristics. In this paper, we present parallelization heuristics for fast community detection using the *Louvain* method as the serial template. The Louvain method is an iterative heuristic for modularity optimization. Originally developed by Blondel *et al.* in 2008, the method has become increasingly popular owing to its ability to detect high modularity community partitions in a fast and memory-efficient manner. However, the method is also inherently sequential, thereby limiting its scalability. Here, we observe certain key properties of this method that present challenges for its parallelization, and consequently propose heuristics that are designed to break the sequential barrier. For evaluation purposes, we implemented our heuristics using OpenMP multithreading, and tested them over real world graphs derived from multiple application domains (e.g., internet, citation, biological). Compared to the serial Louvain implementation, our parallel implementation is able to produce community outputs with a higher modularity for most of the inputs tested, in comparable number of iterations, while providing real speedups of up to $8\times$ using 32 threads. In addition, our parallel implementation was able to exhibit weak scaling properties on up to 32 threads.**

I. Introduction

Community detection, or graph clustering, is becoming pervasive in the data analytics of various fields including (but not limited to) scientific computing, life sciences, social network analysis, and internet applications [1]. As data grows at explosive rates, the need for scalable tools to support fast implementations of complex network analytical functions such as community detection is critical. Given a graph, the problem of community detection is to compute a partitioning of vertices into communities that are closely related within and weakly across communities. Modularity is a metric that can be used to measure the quality of communities detected [2]. Community detection is an NP-Complete problem, but fast heuristics exist. One such heuristic is the Louvain method [3].

Our basis for selecting the Louvain heuristic for parallelization hinges on its increasing popularity within the user community and owing to its strengths in algorithmic and qualitative robustness. With well over 1,000 citations to the original paper (as of this writing), the user base for this method has been rapidly expanding in the last few years. Yet, there is no scalable parallel implementation available for this heuristic. As network sizes continue to grow rapidly into a scale of tens or even hundreds of billions of edges [4], the memory and runtime limits of the serial implementation are likely to be tested. However, parallelization of this inherently serial algorithm can be challenging (as discussed in Sections II and IV).

The parallel solutions presented in this paper (Section V) provide a way to overcome key scalability challenges. In devising our algorithm, we factored in the need to parallelize without compromising the quality of the original serial heuristic and yet be capable of achieving substantial scalability. We also factored in the need for stable solutions across different platforms and programming models. The resulting algorithm, presented in Section V-D, is a combination of heuristics that can be implemented on both shared and distributed memory machines. As demonstrated in our experimental section (Section VI), our implementations provide outputs that have either a higher or comparable modularity to that of the serial method, and is able to reduce the time to solution by factors of up to $8\times$. These observations are supported over a number of real-world networks.

Contributions: The main contributions of this paper are:

i) Introduction of effective heuristics for parallelization of the Louvain algorithm on multithreaded architectures;
ii) Experimental studies using seven real-world networks obtained from varied sources including the DIMACS10 challenge website, University of Florida sparse matrix collection and biological databases;
iii) A simple and clarified derivation of modularity computation that can benefit other researchers exploring community detection algorithms; and
iv) Demonstration of the effectiveness of our parallel heuristics through comparison with the respective serial solutions.

II. Problem statement and notation

Let $G(V, E, \omega)$ be an undirected weighted graph, where V is the set of vertices, E is the set of edges and $\omega(.)$ is a weighting function that maps every edge in E to a non-zero, positive weight[1]. In the input graph, edges that connect

[1]If the graph is unweighted, then we treat every edge to be of weight 1.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.155

a vertex to itself are allowed — i.e., (i, i) can be a valid edge. However, multi-edges are not allowed. Let the adjacency list of i be denoted by $\Gamma(i) = \{j|(i, j) \in E\}$. Let k_i denote the weighted degree of vertex i — i.e., $k_i = \sum_{\forall j \in \Gamma(i)} \omega(i, j)$. We will use n to denote the number of vertices in G; M to denote the *number* of edges in the graph; and m to denote the sum of all edge weights — i.e., $m = \sum_{\forall i \in V} k_i$.

A *community* within graph G represents a (possibly empty[2]) subset of V. In practice, for community detection, we are interested in partitioning the vertex set V into an arbitrary number of *disjoint* non-empty communities, each with an arbitrary size (> 0 and $\leq n$). We call a community with just one element as a *singlet* community. We will use $C(i)$ to denote the community that contains vertex i in a given partitioning of V. We use the term *intra-community edge* to refer to an edge that connects two vertices of the same community. All other edges are referred to as *inter-community edges*. Let $E_{i \to C}$ refer to the set of all edges connecting vertex i to vertices in community C. And let $e_{i \to C}$ denote the sum of the weights of the edges in $E_{i \to C}$ (also referred to as the degree of a community).

$$e_{i \to C} = \sum_{\forall (i,j) \in E_{i \to C}} \omega(i, j) \tag{1}$$

Let a_C denote the sum of the degrees of all the vertices in community C.

$$a_c = \sum_{\forall i \in C} k_i \tag{2}$$

Modularity: Let $P = \{C_1, C_2, \ldots C_k\}$ denote the set of all communities in a given partitioning of the vertex set V in $G(V, E, \omega)$, where $1 \leq k \leq n$. Consequently, the *modularity* (denoted by Q) of the partitioning P is given by the following expression [2]:

$$Q = \frac{1}{2m} \sum_{\forall i \in V} e_{i \to C(i)} - \sum_{\forall C \in P} \left(\frac{a_C}{2m} \cdot \frac{a_C}{2m}\right) \tag{3}$$

Intuitively, modularity is a statistical measure for assessing the quality of a given community-wise partitioning (or equivalently, "clustering"). A "good" clustering method is one that clusters closely related elements (vertices) as part of the same community (or "cluster") while separating weakly related elements into different clusters. In other words, the goal becomes one of maximizing intra-community links while keeping the number of inter-community edges low. This explains the first term in Eqn. (3). However, if the goal is simply to maximize the contribution from intra-community edges, then one could potentially assign all vertices into one community. But such a solution is likely to be meaningless in practice. To overcome this problem, the second term in the Eqn. (3) was introduced. This term represents the fraction of intra-community edges one would expect in an "equivalent" graph (i.e., another graph with the same numbers of vertices and edges, and the same vertex degrees) but with just the edges randomly reconnected.

[2]The notion of empty communities do not have a practical relevance. We have intentionally defined it this way so as to make our later algorithmic descriptions easier. It is guaranteed, however, that all output communities at the end of our algorithm will be non-empty subsets.

Modularity is not the ideal metric for community detection and issues such as resolution limit have been identified [1], [5], and consequently, a few variants of modularity definitions have been devised [5], [6], [7] are available. However, the definition provided in Eqn. (3) continues to be the more widely adopted version in practice, including in the Louvain method [3], and therefore, we will use that definition for this paper.

Community detection: Given $G(V, E, \omega)$, the problem of community detection is to compute a partitioning P of communities that maximizes modularity. This problem has been shown to be NP-Complete [8].

III. The Louvain algorithm

In 2008, Blondel *et al.* presented an algorithm for the community detection [3]. The method, called the *Louvain* method, is an iterative, greedy heuristic capable of producing a hierarchy of communities. The main idea of the algorithm is rather simple and can be summarized as follows: Starting with each vertex in a separate community initially, the algorithm progresses from one iteration to another until the modularity *gain* becomes negligible (as defined by a predefined threshold). Within each *iteration*, the algorithm linearly scans the vertices in an arbitrary but predefined order. For every vertex i, the algorithm examines all its neighboring communities (i.e., the communities containing $i's$ neighbors) and computes the modularity gain that would result if i were to move to each of those neighboring communities from its current community. Once the gains are calculated, the algorithm assigns a neighboring community that would yield the maximum modularity gain, as the new community for i (i.e., new $C(i)$), and updates the corresponding data structures that it maintains for the source and target communities. Alternatively, if all gains turn out to be negative, the vertex stays in its current community. An iteration ends once all vertices are linearly scanned in this fashion. After a phase terminates, the algorithm proceeds to the next *phase* by collapsing all vertices of a community to a single "meta-vertex"; placing an edge from that meta-vertex to itself with an edge weight that is the sum of weights of all the intra-community edges within that community; and placing an edge between two meta-vertices with a weight that is equal to the sum of the weights of all the inter-community edges between the corresponding two communities. The result is a condensed graph $G'(V', E', \omega')$, which then becomes the input to the next phase. Subsequently, multiple phases are carried out until the modularity score converges. Note that each phase represents a coarser level of hierarchy in the community detection process.

At any given iteration, let $\Delta Q_{i \to C(j)}$ denote the modularity gain that would result from moving a vertex i from its current community $C(i)$ to a different community $C(j)$. This term can be calculated in two parts: i) The gain in modularity owing to i leaving $C(i)$ is given by:

$$\begin{aligned} \Delta Q_{C(i)\setminus\{i\}} &= -\frac{e_{i \to C(i)\setminus\{i\}}}{m} + \frac{(a_{C(i)\setminus\{i\}} + k_i)^2 - a^2_{C(i)\setminus\{i\}}}{(2m)^2} \\ &= -\frac{e_{i \to C(i)\setminus\{i\}}}{m} + \frac{2 \cdot k_i \cdot a_{C(i)\setminus\{i\}} + k_i^2}{(2m)^2} \end{aligned} \tag{4}$$

and, ii) The gain in modularity owing to i joining $C(j)$ is

given by:

$$\begin{aligned}\Delta Q_{C(j)\cup\{i\}} &= \frac{e_{i\to C(j)}}{m} - \frac{(a_{C(j)}+k_i)^2 - a_{C(j)}^2}{(2m)^2} \\ &= \frac{e_{i\to C(j)}}{m} - \frac{2\cdot k_i\cdot a_{C(j)} + k_i^2}{(2m)^2} \end{aligned} \quad (5)$$

Therefore,

$$\begin{aligned}\Delta Q_{i\to C(j)} &= \Delta Q_{C(i)\setminus\{i\}} + \Delta Q_{C(j)\cup\{i\}} \\ &= \frac{e_{i\to C(j)} - e_{i\to C(i)\setminus\{i\}}}{m} \\ &\quad + \frac{2\cdot k_i\cdot a_{C(i)\setminus\{i\}} - 2\cdot k_i\cdot a_{C(j)}}{(2m)^2} \end{aligned} \quad (6)$$

Consequently, the new community assignment for i at an iteration is as follows:

$$C(i) = \arg\max_{C(j)} \Delta Q_{i\to C(j)}, \forall j \in \Gamma(i)\cup\{i\} \quad (7)$$

Note that the new $C(i)$ will equal the old $C(i)$ if none of the modularity gains to any of the other neighboring communities evaluates to a positive value. In the implementation, one can maintain several data structures such that each instance of $\Delta Q_{i\to C(j)}$ can be computed in constant time. Consequently, the algorithm's time complexity *per* iteration is $O(m)$. While no upper bound has been established on the number of iterations or on the number of phases, it should be evident that the algorithm is guaranteed to terminate with the use of a cutoff for the modularity gain (because of the modularity being a monotonically increasing function until termination). In practice, the method needs only tens of iterations and fewer phases to terminate on most real world inputs.

IV. Challenges in parallelization

Any attempt at parallelizing the Louvain method should factor in the sequential nature in which the vertices are visited within each iteration of the original algorithm and the impact it has on the overall modularity convergence. Visiting the vertices sequentially gives the advantage of working with the latest information available from all the preceding vertices in this greedy procedure. Furthermore, in the serial algorithm, when a vertex computes its new community assignment (using Eqn.(7)), it does so with the guarantee that no other part of the community structure is concurrently being altered. There can be no such guarantee in *parallel*. In other words, if communities are updated in parallel, it could lead to some interesting situations with an impact on the convergence process.

A. Negative gain scenario

To illustrate the case in point, consider the example scenario illustrated in Figure 1, where two vertices i and j are both connected to a third vertex k with all three of them are in three different communities initially — i.e., $i \in C(i)$, $j \in C(j)$, $k \in C(k)$ s.t. $C(i) \neq C(j) \neq C(k)$. If both vertices i and j evaluate the possibility of moving to $C(k)$ independently, using Eqn.(6), then from each of their perspectives, their *predicted* value for the new modularity is $Q_{old} + \Delta Q_{i\to C(k)}$ and $Q_{old} + \Delta Q_{j\to C(k)}$, respectively. However, if both i and j decide to move to $C(k)$ in parallel, then the *actual* value for the new modularity will be $Q_{old} + \Delta Q_{\{i,j\}\to C(k)}$, where:

$$\begin{aligned}\Delta Q_{\{i,j\}\to C(k)} &= \Delta Q_{i\to C(k)} + \Delta Q_{j\to C(k)} \\ &\quad + \frac{\omega(i,j)}{m} - \frac{2\cdot k_i\cdot k_j}{(2m)^2} \end{aligned} \quad (8)$$

If $(i,j) \notin E$, $\omega(i,j) = 0$, implying:

$$\begin{aligned}\Delta Q_{\{i,j\}\to C(k)} &= \Delta Q_{i\to C(k)} + \Delta Q_{j\to C(k)} \\ &\quad - \frac{2\cdot k_i\cdot k_j}{(2m)^2} \\ &\leq \Delta Q_{i\to C(k)} + \Delta Q_{j\to C(k)} \end{aligned} \quad (9)$$

Furthermore, if $\Delta Q_{i\to C(k)} + \Delta Q_{j\to C(k)} < \frac{2\cdot k_i\cdot k_j}{(2m)^2}$

$$\Rightarrow \quad \Delta Q_{\{i,j\}\to C(k)} < 0 \quad (10)$$

On the other hand, if $\frac{\omega(i,j)}{m} > \frac{2\cdot k_i\cdot k_j}{(2m)^2}$ (can be true only if $(i,j) \in E$), then:

$$\Delta Q_{\{i,j\}\to C(k)} > \Delta Q_{i\to C(k)} + \Delta Q_{j\to C(k)} \quad (11)$$

This is because $\Delta Q_{i\to C(k)} > 0$ and $\Delta Q_{j\to C(k)} > 0$; the latter two inequalities follow from the fact that i and j chose to move to $C(k)$. Note that if this happens, then parallel version could potentially surpass the serial version toward modularity convergence. Inequalities (9-11) imply the following lemma:

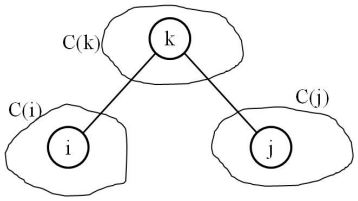


Fig. 1. Illustration of the negative gain scenario using an example of three vertices (Lemma 1).

Lemma 1: At any given iteration of the Louvain algorithm, if community updates for vertices are performed in parallel, then the net modularity gain achieved cannot be guaranteed to be always positive.

The above lemma has a direct implication on the convergence property of the Louvain method, one way or another. Pessimistically speaking, if the net modularity gain can become negative between consecutive iterations of the algorithm, then there is no theoretical guarantee that the algorithm will terminate. Even if the chances of non-termination turn out to be bleak, it could potentially slow down the rate at which the algorithm progresses toward a solution, causing more number of iterations. For this reason, the *number of iterations* that the algorithm takes to converge toward the solution and the *quality of the solution* relative to the serial algorithm's can be good indicators of the effectiveness of a parallel strategy. Note that the above example with three vertices can be extended to scenarios where multiple unrelated vertices are trying to enter a community at its periphery without mutual knowledge.

B. Swap and local maxima scenarios

There is also another type of scenario that could impede the progression of the parallel algorithm toward a solution. Consider a simple example where two vertices i and j connected by an edge $(i, j) \in E$ s.t., $C(i) = \{i\}$ and $C(j) = \{j\}$. In the interest of increasing modularity, if the two vertices make a decision to move to each other's community concurrently, then such an update could potentially result in both vertices simply swapping their community assignments without achieving any modularity gain. This could also happen in a more generalized setting, where subsets of vertices between two different communities swap their community assignments, each unaware of the other's intent to also migrate. Swap conditions or cyclic migration patterns could lead to a deadlock which need to be detected and surpassed in parallel. Note that this is not a problem with the serial algorithm, as only one decision is made at any given point of time.

A parallel algorithm also runs the risk of settling on a locally optimal solution when there is a better solution available. This could happen even in serial; in parallel such scenarios may arise if a single community gets partitioned into equally weighted sub-communities, in which there is no incentive for any individual vertex to merge with any of the other sub-communities; and yet, if all vertices from each of the sub-communities were to merge together to form a single community the net modularity gain could be positive. An example of this case will be shown later in Section V-C. Getting stuck in a locally optimal solution, however, can be resolved when the algorithm progresses to subsequent phases (as will be described in Section V-C).

V. Parallel heuristics

In this section, we present our ideas to tackle the challenges outlined above in parallelizing the Louvain heuristic community detection.

A. Coloring

Vertex coloring could provide an effective way to address some of the cases presented in both scenarios (negative gain and swap) in practice. It should be easy to see that using distance-1 coloring to partition the vertices into colors prior to the processing would prevent the vertex-to-vertex swap scenarios explained under Section IV-B. In this scheme, vertices of the same color are processed in parallel, and this is equivalent of guaranteeing that no two adjacent vertices will be processed concurrently. However, distance-1 coloring may not be adequate to address the other potential complications that may arise during parallelization (see Section IV-A).

Corollary 2: Applying and processing in parallel the vertices by distance-1 coloring does *not* necessarily preclude the possibility of negative modularity gains between iterations.

Proof: Follows directly from the three vertex example case presented for Lemma 1. ■

In fact the same result can be shown for even an arbitrary distance-k coloring scheme.

Lemma 3: Applying and processing the vertices by a distance-k coloring scheme, where $k > 1$, still does *not* preclude the possibility of negative modularity gains between iterations.

Proof: Let i and j be two vertices that are separated by a shortest path distance of at least $k + 1$ hops in the graph. Furthermore, let the two vertices belong to two different communities: $C(i) \neq C(j)$. This implies that under a distance-k coloring scheme, it is possible for vertices i and j to be processed concurrently. Even though the vertices are separated by at least $k + 1$ hops, there are two cases in which it is still possible for them to concurrently update the same target community. The cases are illustrated in Figure 2. Let i' and j' denote some neighbor of i and j, respectively. Case a) shows the possibility where there could exist $i' \in \Gamma(i)$ and $j' \in \Gamma(j)$ such that $C(i') = C(j')$, and vertices i and j decide to move to $C(i')$ (as dictated by Eqn.7). Case b) shows the possibility where there could exist $i' \in \Gamma(i)$ such that $C(i') = C(j)$, and vertex i decides to move to $C(j)$ and vertex j decides to stay. In both cases, the vertices i and j would update the same target community in a concurrent fashion, and by Lemma 1, that could potentially lead to negative modularity gains. ■

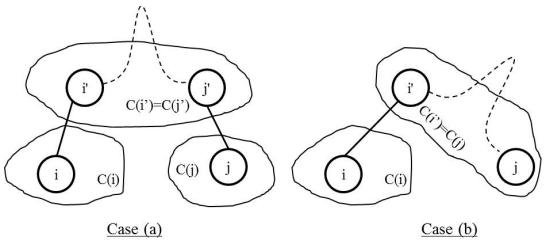


Fig. 2. Illustration for showing the different cases for which even an arbitrary distance-k coloring used for processing the vertices in parallel may still not be sufficient to prevent negative modularity gains (Lemma 3). In all three cases, vertices i and j are separated by a shortest path distance of at least $k + 1$.

Despite these lack of guarantees for a positive modularity gain between iterations, coloring still could be effective as a heuristic in practice. The cases outlined above in the proof for the Lemma 3 are expected to be rare, given the number of conditions that need to be met. The disadvantage of coloring, however, is that it may reduce the degree of parallelism. Note also that the coloring could add an overhead, but it is a preprocessing step that needs to be performed only once prior to each phase, and there are efficient coloring implementations available [9].

B. The vertex following heuristic

In this section, we will layout a particular property of the *serial* Louvain algorithm in the way it treats vertices with single neighbors, and devise a heuristic around it. For the lemma and the proof below, we will assume the version of Louvain algorithm which continues with iterations within a phase, until the communities stop changing.

Lemma 4: Given an input graph $G(V, E, \omega)$, let i and j be two vertices in E such that $(i, j) \in E$ and $|\Gamma(i)| = 1$. Then, the final solution should have $C(i) = C(j)$ — i.e., i should be part of the same community as j.

Proof: There are two parts to this proof. First, we will show that vertex i will join $C(j)$ in the first iteration (*Claim 1*). Second, we will show that if at any iteration, say r ($r \geq 1$), j decides to leave i for a newer community, then either in the current (r) or next ($r + 1$) iteration, i will again become part of $j's$ community (*Claim 2*).

Claim I) At the start of the first iteration, $C(i) = \{i\}$ and $C(j) = \{j\}$. There are two sub-cases:
Ia) i is processed after j (i.e., $i > j$); or
Ib) j is processed after i (i.e., $i < j$).

Sub-case Ia) Since j is the only neighbor of i, vertex i only has two choices: i) to stay in a separate community by itself; or ii) to join $j's$ community. The choice will be determined by calculating the modularity gain to migrate to $C(j)$ as follows:

$$\begin{aligned}\Delta Q_{i\rightarrow C(j)} &= \frac{\omega(i,j)}{m} + \frac{-2\cdot k_i \cdot a_{C(j)}}{(2m)^2} \\ &= \frac{\omega(i,j)}{m} - \frac{\omega(i,j)\cdot a_{C(j)}}{2m^2} \ (\because \Gamma(i) = \{j\})\end{aligned}$$

For i to decide *against* moving to $C(j)$:

$$\begin{aligned}\Delta Q_{i\rightarrow C(j)} &\leq 0 \\ \Rightarrow \frac{\omega(i,j)}{m} - \frac{\omega(i,j)\cdot a_{C(j)}}{2m^2} &\leq 0 \\ \Rightarrow \frac{\omega(i,j)}{(2m)^2}(2m - a_{C(j)}) &\leq 0 \qquad (12) \\ \Rightarrow 2m - a_{C(j)} &\leq 0 \ (\because \frac{\omega(i,j)}{(2m)^2} > 0) \\ \Rightarrow 2m &\leq a_{C(j)}\end{aligned}$$

But this is *not* possible because $a_{C(j)} \leq 2m$ for any community (by the definition in Eqn.2) and in this case, since $i \notin C(j)$, $a_{C(j)} \leq (2m - \omega(i,j)) < 2m$. This implies that i will have no choice but to move to $C(j)$ in the first iteration.

Sub-case Ib) It may happen that vertex i is processed before j in this iteration, in which case, i will first move to $C(j)$ (by the result of sub-case (1)). However, it is possible that later when j is processed, it decides to move out of its current community (same as $C(i)$) to a different community, say $C(k)$. This move will imply the following updates: $C(i) = C(i)\backslash\{j\}$ and $C(j) = C(k) = C(k) \cup \{j\}$. Consequently, $C(i) \neq C(j)$ at the end of this iteration. If this happens, then this reduces to the case addressed in Claim II (see below).

Claim II) Let at some iteration r ($r \geq 1$), vertex j move out of the community containing i — i.e., $C(i) \neq C(j)$. Subsequently, when vertex i is processed (in iteration r or $r+1$), i will evaluate its modularity gain to the new $C(j)$, and move to $C(j)$ if and only if $\Delta Q_{i\rightarrow C(j)} > 0$.

$$\begin{aligned}\Delta Q_{i\rightarrow C(j)} &= \frac{\omega(i,j)}{m} + \frac{2\cdot k_i \cdot a_{C(i)\backslash\{i\}} - 2\cdot k_i \cdot a_{C(j)}}{(2m)^2} \\ &= \frac{\omega(i,j)}{2m^2}\left\{2m + a_{C(i)\backslash\{i\}} - a_{C(j)}\right\} \\ &\qquad (\because k_i = \omega(i,j)) \\ &\geq \frac{\omega(i,j)}{2m^2}\left\{2m - a_{C(j)}\right\} \ (\because a_{C(i)\backslash\{i\}} \geq 0)\end{aligned}$$

If i were to decide *against* moving to $C(j)$, $\Delta Q_{i\rightarrow C(j)} \leq 0$. Given that the above inequality is a lower bound for $\Delta Q_{i\rightarrow C(j)}$:

$$\Rightarrow \frac{\omega(i,j)}{2m^2}\left\{2m - a_{C(j)}\right\} \leq 0 \qquad (13)$$

Inequality.(13) is in the same form as Inequality.(12). Consequently, i is guaranteed to move to $C(j)$. ■

We refer to this as the *vertex following (VF) rule* and it applies only to vertices which have a single neighboring vertex. Since this rule says that regardless of the rest of the graph, a vertex with only a single neighbor will also follow that neighbor in its community assignment, there is no need to consider making decisions on single neighbor vertices during the algorithm's iterations. However, we should not completely ignore these vertices either as their links may dictate the community assignments for the other vertices. Therefore, we preprocess the input such that all vertices satisfying the single neighbor property are merged as part of their respective neighbor's community initially. More specifically, let i be a single neighbor vertex and its neighbor be j. Then, we remove vertex i from the graph, and replace j with a new vertex j', such that $\Gamma(j') = \{\Gamma(j) \setminus \{i\}\} \cup \{j'\}$ and $\omega(j', j') = \omega(i,j)$ if $(j,j) \notin E$; and $\omega(j', j') = \omega(j,j) + \omega(i,j)$ otherwise. Note that if for some single neighbor vertex i, its neighbor j is also a single neighbor, then the i is merged with j only if $i < j$ (as a convention).

This preprocessing not only could help reduce the number of vertices that need to be considered during each iteration, but it also allows the vertices that contain multiple neighbors (which represent the hubs in the networks) be the main drivers of community migration decisions. This is more important under a parallel setting because if the single neighbor vertices were retained in the network the hub nodes may tend to gravitate temporarily toward one of their single neighbor mates, thereby delaying progression of solution or getting stuck in a local maxima.

C. The minimum label heuristic

Section IV-B elaborated on the possibilities of swapping conditions that may delay the parallel algorithm's convergence to a solution. In this section we present a heuristic designed to address some of these cases. Let us consider the simple case of two vertices i and j outlined in Section IV-B. Here both vertices are initially in communities of size one, and a decision in favor of merging at any given iteration will lead them to simply swap their respective communities without resulting in any net modularity gain. This is outlined in the Case 1a of Figure 3. Such a swap can be easily prevented by introducing a labeling scheme where it can be enforced that only one of them move to other's community. More specifically, let the communities at any given stage of the algorithm be labeled numerically (in an arbitrary order). We will use the notation $\ell(C)$ to denote the label of a community C. Then the heuristic is as follows:

The singlet minimum label heuristic: In the parallel algorithm, at any given iteration, if a vertex i which is in a community by itself (i.e., $C(i) = \{i\}$), decides (in the interest of modularity gain) to move to another community $C(j)$ which also contains only one vertex j, then that move will be performed *only if* $\ell(C(j)) < \ell(C(i))$.

This heuristic prevents single-member communities from swapping their vertices as that will *not* result in a net modularity gain.

The above heuristic can be generalized to many other cases of swapping or local maxima. For instance, let us consider the 4-clique of $\{i_4, i_5, i_6, i_7\}$ shown in Figure 3: case 2, assuming that each vertex is in its own individual community to start

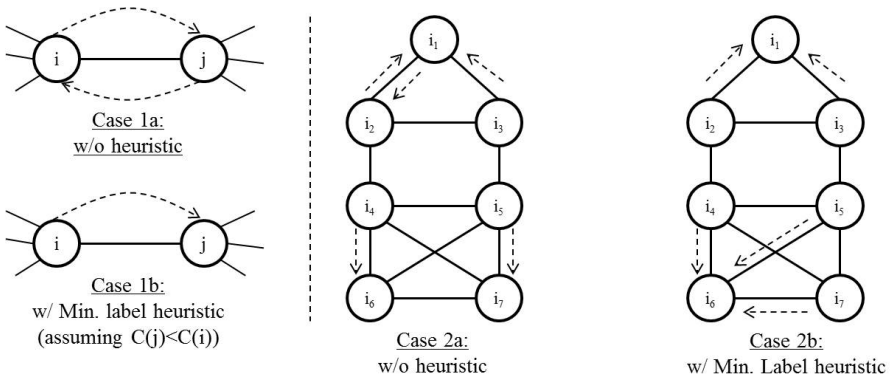


Fig. 3. Examples of cases which can be handled by using the minimum degree/labeling heuristic. The dotted arrows point to the direction of the vertex migration. Case 1 shows a scenario of vertex swap between two communities. Case 2) shows the evolution of two different communities $\{i_1, i_2, i_3\}$ and $\{i_4, i_5, i_6, i_7\}$. Without the application of any heuristic (Case 2b), the algorithm may either form partial communities (e.g., $\{i_1\}, \{i_2, i_3\}$) or may settle on a local maxima (e.g., $\{i_4, i_6\}, \{i_5, i_7\}$). Whereas the use of a minimum label heuristic could help the communities converge to the final solutions faster (as shown in Case 2b).

with. Here, in the absence of an appropriate heuristic there is a chance that the algorithm would settle on a local maxima. For instance, maximum modularity gains can be achieved at vertex i_4 by either moving to $C(i_6)$ or $C(i_7)$, and similarly for vertex i_5. However, if i_4 moves to $C(i_6)$ and i_5 to $C(i_7)$, then the resulting solution $\{i_4, i_6\}, \{i_5, i_7\}$ (shown in case 2a of Figure 3) will represent a local maxima from which the algorithm may not proceed in the current phase. This is because, once these partial communities form, there is no incentive for i_4 or i_6 to individually move to the community containing $\{i_5, i_7\}$, without each other's company. This is a limitation imposed by the Louvain heuristic, which makes decisions at the vertex level. However, if we label and treat the communities in a certain way then such local maxima situations can be avoided.

The generalized minimum label heuristic: In the parallel algorithm, at any given iteration, if a vertex i has *multiple* neighboring communities yielding the maximum modularity gain, then that vertex which has the minimum label among them will be selected as i's destination community.

In the example for Figure 3:case 2, vertices i_6 and i_7 will both yield the maximum modularity gain for vertices i_4 and i_5. However, using the above minimum label heuristic, all three vertices $\{i_4, i_5, i_7\}$ will migrate to $C(i_6)$, while i_6 stays in $C(i_6)$ — i.e., assuming $\ell(C(i_4)) < \ell(C(i_5)) < \ell(C(i_6)) < \ell(C(i_7))$.

While we found the above heuristics to be generally effective in mitigating several swapping and local maxima situations (see Section VI), they do not provide a protection against *all* such cases. While swap situations may delay convergence, it can be shown that they can never lead to nontermination of the algorithm due to the use of a predefined cutoff for the net modularity gain. Similarly, the local maxima situations explained generally get resolved in subsequent phases, where the representation of the individual sub-communities as meta-vertices is likely to lead them to merge with one another forming the containing communities eventually in the output. Due to the lack of space, the proofs for these assertions are not provided here in this paper.

D. Parallel algorithm

Our parallel algorithm has the following major steps:

1) Preprocessing (Optional): Apply the vertex following heuristic by merging all single neighbor vertices as part of their respective neighboring vertices initially (as explained in Section V-B).
2) Preprocessing: Label the vertices from $1 \ldots n$ in any particular order of choice.
3) Preprocessing (Optional): Color the input vertices using distance-k coloring. In this paper, we used distance-1 coloring. For coloring, we used the parallel implementation from [9].
4) Phases: Execute phases one at a time as per Algorithm 1. Within each phase, the algorithm runs multiple iterations of the Louvain algorithm until there is no longer any appreciable modularity gain between successive iterations. The iterations are the main parallel steps, and the algorithm for each iteration is also shown within Algorithm 1.
5) Input transformation: Between two successive phases, the community assignment output of the completed phase is used to construct the input graph for the next phase. This is done by representing all communities of the completed phase as "vertices" and accordingly introducing edges, identical to the manner in which it is done in the serial algorithm. Note that one can think of applying all the above three preprocessing steps to the inputs going into each of the phases, starting from the second phase, as well. However, due to the high levels of graph contraction obtained in the transformation step, we typically found it unnecessary.

While the pseudocode shown in Algorithm 1 is more representative of a shared memory multicore algorithm, the same set of heuristics and ideas carry over to distributed memory parallelism except for the implementation.

We note here that the above parallel algorithm, with the exception of coloring heuristic, is stable in that it always produces the same output regardless of the number of processors used. This is owing to the fact that at each iteration the decisions made by every vertex are based on the state of communities from the previous iteration (proof trivial). When coloring is

Algorithm 1 The parallel Louvain algorithm (a single phase).

1: **procedure** PARALLEL LOUVAIN($G(V, E, \omega), C$)
2: `Initialization`
3: **for each** $i \in V$ in `parallel` **do**
4: $C(i) \leftarrow \{i\}$; $\ell(C(i)) \leftarrow i$
5: $C^i_{int} \leftarrow 0$ ▷ counter for the #intra-community edges due to i
6: **for each** $j \in \Gamma(i)$ **do**
7: $C^i_{tot} \leftarrow C^i_{tot} + \omega(i, j)$
8: $Q_C \leftarrow 0$ ▷ Current modularity
9: $Q_P \leftarrow -\infty$ ▷ Previous modularity
10: **while** `true` **do** ▷ Iterate until modularity gain becomes less than a user specified threshold.
11: ▷ `Stage-1`: For each vertex, compute the modularity gain from moving to a neighboring cluster.
12: **for each** $i \in V$ in `parallel` **do**
13: $C_{old} \leftarrow C(i)$
14: $N_i \leftarrow C(i)$ ▷ Neighboring communities of i
15: **for each** $j \in \Gamma(i)$ **do**
16: $N_i \leftarrow N_i \cup C_j$
17: $maxGain \leftarrow 0$
18: $C_{new} \leftarrow C_{old}$
19: **for each** $c \in N_i$ in `parallel` **do**
20: $curGain \leftarrow$ Calculate $\Delta Q_{i \to c}$
21: **if** $((curGain > maxGain)$ or $(curGain = maxGain$ and $\ell(c) < \ell(C_{new}))$ **then** ▷ Minimum label heuristic
22: $maxGain \leftarrow curGain$
23: $C_{new} \leftarrow c$
24: **if** $maxGain > 0$ **then**
25: $C_{old} \leftarrow C_{old} \setminus \{i\}$
26: $C_{new} \leftarrow C_{new} \cup \{i\}$
27: ▷ `Stage-2`: Compute the net modularity for this iteration.
28: **for each** $c \in C$ `AND` $c \neq \emptyset$ in `parallel` **do**
29: $C^c_{int} \leftarrow 0$
30: $C^c_{tot} \leftarrow 0$
31: **for each** $(i, j) \in E$ in `parallel` **do**
32: **if** $C(i) = C(j)$ **then**
33: $C^i_{int} = C^i_{int} + \omega(i, j)$
34: $C^i_{tot} = C^i_{tot} + \omega(i, j)$
35: **else**
36: $C^i_{tot} = C^i_{tot} + \omega(i, j)$
37: $C^j_{tot} = C^j_{tot} + \omega(i, j)$
38: $e_{xx} \leftarrow 0$
39: $a^2_x \leftarrow 0$
40: **for each** $c \in C$ `AND` $c \neq \emptyset$ in `parallel` **do**
41: $e_{xx} += C^c_{int}$
42: $a^2_x += (C^c_{tot})^2$
43: $Q_C = \frac{e_{xx}}{m} - \frac{a^2_x}{(2m)^2}$
44: **if** $|\frac{Q_C - Q_P}{Q_P}| < \theta$ **then** ▷ θ is a user specified threshold.
45: `break` ▷ Phase termination
46: **else**
47: $Q_P \leftarrow Q_C$

applied, the use of differing number of threads within a given iteration could potentially vary the order in which decisions are made, thereby leading to potential variations in the output modularities. In our experiments, we found the magnitudes of such variations to be negligible.

E. Implementation

We implemented our parallel heuristics in C++/OpenMP. It is to be noted that the heuristics themselves are agnostic to the underlying parallel architecture. There are a few implementation level variations to the algorithm presented in Algorithm 1. In Stage-2, the modularity calculation happens in steps $38 - 43$. The steps from $27 - 37$ calculates the intra- and inter-community edge counts required for the modularity calculation. In our actual implementation we do not explicitly execute these steps. Instead we aggregate these values during Stage-1, as net modularity gains are being calculated. This saves significant rework. Secondly, steps $25-26$ show the update for the source and target communities for each vertex i. We implemented these updates using intrinsic atomic operations `__sync_fetch_and_add()` and `__sync_fetch_and_sub()`.

F. Analysis

Within each parallel iteration of Algorithm 1, the vertices are scanned in parallel, and for every vertex their vertex neighborhood is scanned first to curate the set of distinct neighboring communities (steps $14 - 16$). Subsequently, the main step of modularity gain calculation is performed only for each distinct neighboring community (steps $19 - 23$), which is equal to vertex degree initially but is expected to rapidly reduce as the iterations progress. Consequently, the worst-case runtime complexity *per* iteration is $O(\max\{\frac{M+n\cdot\lambda}{p}, \lambda_{max}\})$, where p denote the number of processing cores, λ is the average (unweighted) degree of a vertex and λ_{max} is the maximum (unweighted) degree of a vertex. The space complexity is linear in the input for shared memory implementation (i.e., $O(m + n)$), whereas $O(\frac{m+n}{p})$ for the distributed memory implementation.

VI. EXPERIMENTAL EVALUATION

A. Experimental setup

The test platform for our experiments is an Intel Xeon X7560 server with four sockets and 256 GB of memory. Each socket is equipped with eight cores running at 2.266 GHz, leading to a total of 32 cores. The system is equipped with 32 KB of L1 and 256 KB L2 caches per core, and 24 MB of cache per socket. Each socket has 64 GB of DDR3 memory with a peak bandwidth of 34.1 GB per second. Further details of the machine are available in [10]. The software was compiled with Intel 11.1 compilers using `-fast` option. We also enabled non-uniform memory distribution using `numactl` command and enabled thread binding by using `KMP_AFFINITY` set to `scatter`. This option places the threads across the system as evenly as possible. In all the experiments, we placed one thread per core.

We tested our heuristics on 7 different real world input graphs, the statistics of which are summarized in Table I. With the exception of inputs labeled "MG1" and "MG2", all other inputs were downloaded from the DIMACS10 challenge website [4], [11], and the University of Florida sparse matrix collection [12]. "MG1" and "MG2" are graphs constructed for two different ocean metagenomics data, using the construction procedure described in [13].

Each of the real world inputs were tested using multiple variants of our implementation that use different combination of heuristics, to allow for the evaluation of each heuristic. These variants are as follows:

- **baseline:** represents our parallel implementation with only the Minimum Labeling (ML) heuristic;

TABLE I. INPUT STATISTICS FOR THE REAL WORLD NETWORKS USED IN OUR EXPERIMENTAL STUDY. "RSD" STANDS FOR THE RELATIVE STANDARD DEVIATION OF THE GRAPH'S NODE (UNWEIGHTED) DEGREES, AND IS GIVEN BY THE RATIO BETWEEN THE STANDARD DEVIATION OF THE DEGREE AND THE MEAN.

Input graph	No. vertices (n)	No. edges (M)	Degree statistics (λ) max.	avg.	RSD
CNR	325,557	2,738,970	18,236	16.826	13.024
coPapersDBLP	540,486	15,245,729	3,299	56.414	1.174
Channel	4,802,000	42,681,372	18	17.776	0.061
Europe-osm	50,912,018	54,054,660	13	2.123	0.225
MG1	1,280,000	102,268,735	148,155	159.794	2.311
uk-2002	18,520,486	261,787,258	194,955	28.270	5.124
MG2	11,005,829	674,142,381	5,466	122.506	2.370

TABLE II. DISTANCE-1 COLORING STATISTICS FOR THE ORIGINAL INPUT GRAPHS. "COLOR SIZE" IS MEASURED IN THE NUMBER OF VERTICES BELONGING TO THE SAME COLOR.

Input graph	No. colors	Color size statistics max.	min.	avg.	RSD	Time (in sec) to color using 16 threads
CNR	85	183,614	1	3,830	5.786	0.025
coPapersDBLP	336	59,319	1	1,609	3.477	0.094
Channel	155	1,117,935	154	436,545	1.236	0.299
Europe-osm	5	24,605,946	48,772	12,727,964	1.046	0.920
MG1	799	290,969	1	1,602	7.780	1.129
uk-2002	943	10,684,895	1	19,640	18.876	0.976
MG2	2,143	2,289,843	1	5,136	12.748	9.909

- **baseline+VF:** represents the baseline implementation with the application of the Vertex Following (VF) heuristic in a preprocessing step;
- **baseline+Color:** represents the baseline implementation with the application of the coloring heuristic applied in a preprocessing step;
- **baseline+VF+Color:** represents the baseline implementation with the application of both the VF and coloring heuristics (in that order).

B. Performance evaluation

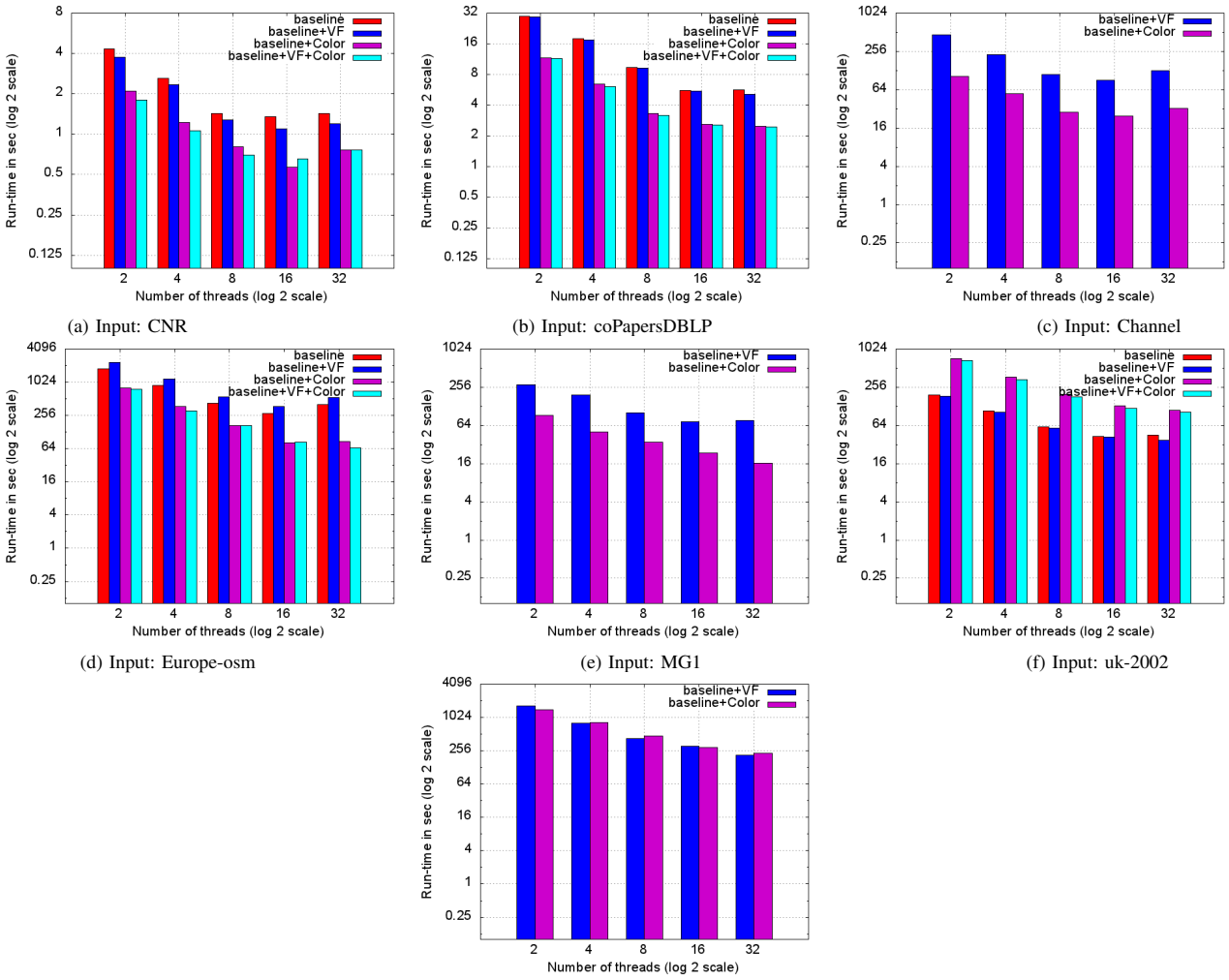
Figures 4a-g show the parallel runtimes of the different implementation variants listed above for each of the test inputs, as a function of the number of cores (2 through 32). Figures 5a-g show the evolution of modularity from the first iteration of the first phase to the last iteration of the last phase, for each implementation variant.

baseline+VF: The primary improvement expected out of the VF heuristic over the baseline implementation is the reduction in runtime for each iteration, due to the reduced number of vertices processed. However, the effectiveness of the VF heuristic is also tied to the number of single degree vertices in the original input graph. Among the inputs tested, we applied the VF heuristic to the following: {CNR,coPapersDBLP,Channel,Europe-osm,uk-2002}. There was no need to apply the heuristic on MG1 and MG2 inputs because their single degree vertices ($\sim$10%) had already been pruned off when the graphs were generated[3]. As can be observed from the runtime charts in Figure 4, the runtime improvement is most pronounced in CNR and uk-2002. This can be attributed to their respective skewed degree distributions, as confirmed by their high RSD values in Table I. The Channel input is devoid of single degree vertices and therefore its baseline case is identical to its baseline+VF (shown). The baseline+VF results for coPapersDBLP and Europe-osm are interesting in that, even though the runtime per iteration reduced with the application of VF, the change in the input graph adversely led to an increase in the overall number of iterations for convergence, thereby increasing time to completion. This is confirmed by the respective modularity curves in Figure 5b,d. This is in contrast to inputs CNR and uk-2002, where the application of VF also results in faster modularity gains.

baseline+Color: The primary design intent of coloring is to reduce the number of iterations required to converge on a solution, and in the process, also reduce the time to solution. However, note that the cost of coloring is reduced parallelism within each iteration; more specifically, the presence of a numerous small color sets could result in an under-utilization of thread-level parallelism. In our experimental results, for all inputs except uk-2002, we found coloring to be highly effective in reducing both the number of iterations *and* the overall time to solution. These results are shown by the *baseline+Color* plots in Figures 4 and 5. These improvements were evident in CNR, coPapersDBLP, Channel, Europe-osm and MG1, with the run-time reduction anywhere from $\sim 2\times$ (CNR) to $4.6\times$ (Europe-osm, 32 threads). In contrast, the run-time improvements were either negligible in the case of MG2 or negative in the case of uk-2002. These observations correlate with the skew in color size distributions of the respective graphs, as shown by the coloring statistics for each input in Table II. For instance, the relatively higher values in the Relative Standard Deviation (RSD) of the color sizes (Table II) for the inputs uk-2002 and MG2 indicate the presence of a large number of small color sizes that could result in thread under-utilization. We plan to improve the color size distribution by introducing randomization in the coloring algorithm, towards a better load balanced distribution. For all inputs, however, the overall number of iterations required for modularity convergence is significantly reduced because of coloring (see Figure 5).

We enable coloring only for the first and the last phases. For the first phase, with coloring enabled, we set the threshold value to a higher value (10^{-2}). For the rest of the phases, we use a default threshold of 10^{-6}. Note that the value of

[3]As a result, there are no "baseline" results to show on MG1 and MG2.



(a) Input: CNR

(b) Input: coPapersDBLP

(c) Input: Channel

(d) Input: Europe-osm

(e) Input: MG1

(f) Input: uk-2002

(g) Input: MG2

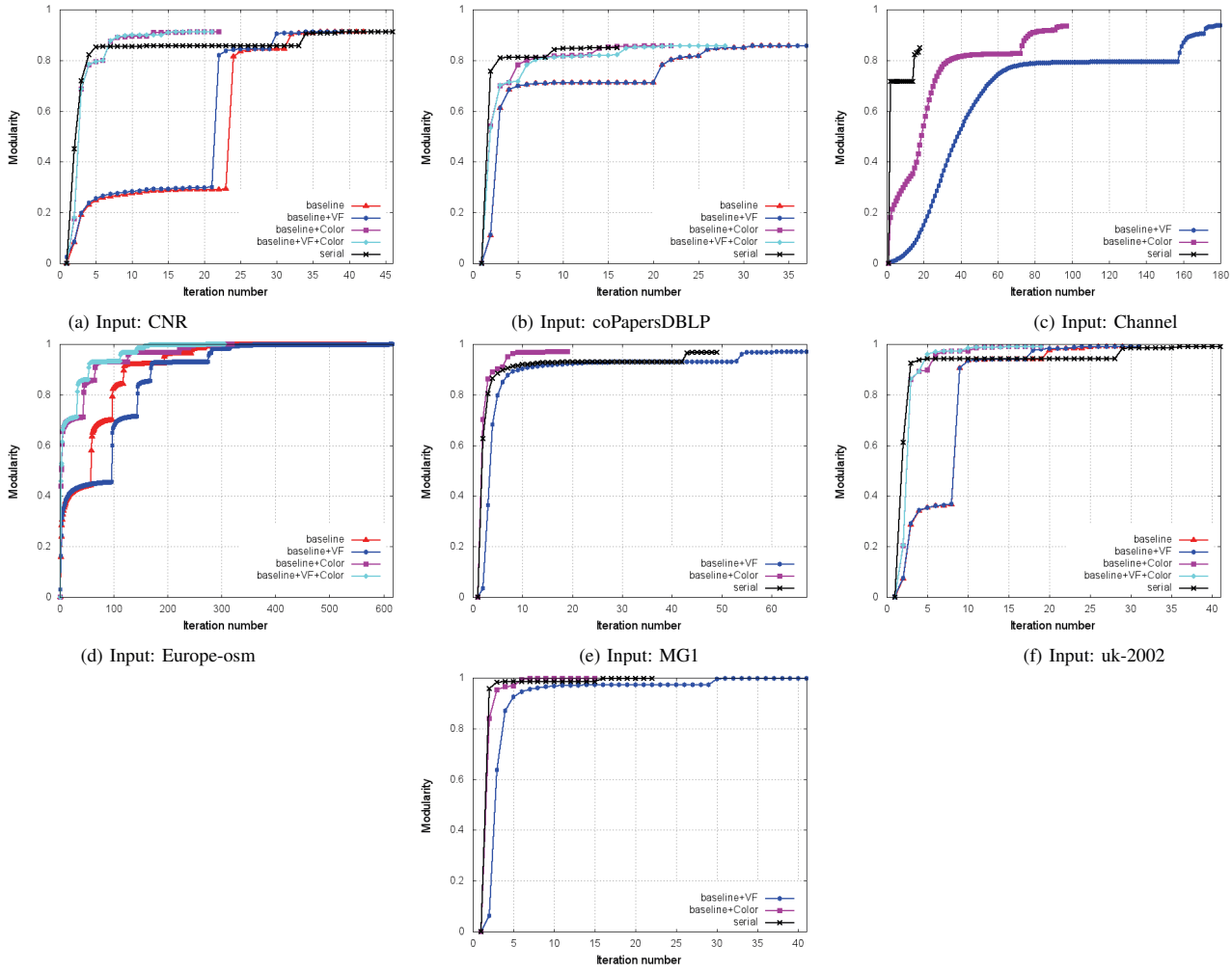
Fig. 4. The charts show the parallel runtimes of the different heuristic combinations as a function of the number of threads (cores) used.

threshold determines the minimum value in modularity gain before the execution is terminated. Since coloring considers independent nodes to maximize modularity, we reason that the gain in modularity will be higher than when coloring is not used.

Effect of both VF and coloring: We also studied the effect of applying both VF and coloring heuristics, by preprocessing in two steps — i.e., VF followed by coloring. This was performed for the inputs CNR, coPapersDBLP, Europe-osm and uk-2002, which had single degree vertices. The results are shown by the baseline+VF+Color plots in Figures 4 and 5. The results show the two heuristics complement one another by resulting in additive net gains.

Scaling and run-time results: Figure 4 also shows the run-time scaling of our parallel implementation variants as a function of the number of threads. A weak scaling trend is apparent — for the smaller inputs such as CNR the implementation linear scaling is observed until 8 threads, and for the larger inputs (e.g., Europe-osm, uk-2002) the scaling extends to 32 threads. However, graph size is not the only factor to affect scaling; other factors such as the degree distribution can also be influential. For instance, the best relative speedup was observed for the medium size Europe-osm input (e.g., on 16 threads, the baseline+Color achieved a 9.8x improvement in run-time over the corresponding 2-thread run).

In all our runs, we observed the total run-time of an execution to be dominated by the run-time for Phase 1. In most cases, Phase 1 took more than 99% of the total run-time (detailed phase-wise breakdown not shown due to lack of space). For instance, for the MG2 input (baseline run), Phase 1 took 1647 seconds out of the total 1650 seconds on two threads, and the contributions from Phase 1 were of similar magnitude or more for other core counts too. This is expected as after Phase 1, the graph condenses significantly. More interestingly, we observed a significant variance in the iteration-wise runtimes within Phase 1. This is counter-intuitive because the fixed number of vertices and edges within a phase seem to suggest that the total work should also stay the same



(a) Input: CNR

(b) Input: coPapersDBLP

(c) Input: Channel

(d) Input: Europe-osm

(e) Input: MG1

(f) Input: uk-2002

(g) Input: MG2

Fig. 5. Charts showing the evolution of modularity from the first iteration of the first phase to the last iteration of the last phase, for each of our parallel implementation variants using different combination of heuristics. The phase transitions are marked by the steep climbs following plateaus that are indicative of convergence within a phase. In addition to our results, we also show the serial Louvain implementation's results to enable a comparison on the highest modularity achieved and the number of iterations taken to reach to that value.

across the iterations of the phase. The variance is because of the drastic reduction in the number of communities as the iterations progress within the first phase (plot not shown). The number of communities matter because within each iteration of our parallel algorithm, the number of modularity gain calculations per vertex is proportional to the number of distinct neighboring communities of the vertex.

Comparison to serial Louvain: We also comparatively evaluated the performance of our parallel implementations proposed in this paper against the publicly available serial Louvain distribution [14]. For all the inputs tested, the final modularities achieved by all our parallel versions agree up to a minimum of three decimal places. In some cases, variance beyond the third decimal places was observed in the final output modularities when coloring was applied across different thread counts. Table III compares the final modularities achieved by our parallel implementation versus the best modularities achieved by the serial Louvain code for each input. The table also shows the best run-time achieved by our parallel implementation against the serial run-time. For the inputs CNR and uk-2002, the serial Louvain outputs a higher modularity than our parallel implementation, although the differences are marginal, with the modularity figures agreeing up to the third decimal place. On the other hand, our parallel implementation delivers a higher modularity than the serial method for inputs coPapersDBLP, Channel and MG1. In particular, the improvements are most pronounced for coPapersDBLP (+0.009) and Channel (+0.085). As for the run-times, we observe that our parallel implementation delivers real speedups in the range of $3.1\times$ to $8.2\times$ over the serial implementation, for all inputs except Europe-osm and Channel. For Europe-osm, the serial implementation was crashing, for reasons we are yet to identify. For the Channel input, observe from Table I that the degree distribution is highly uniform. This could cause

TABLE III. COMPARISON OF THE BEST MODULARITIES AND RUN-TIMES ACHIEVED BY OUR PARALLEL IMPLEMENTATION VERSUS THE CORRESPONDING VALUES ACHIEVED BY THE SERIAL LOUVAIN IMPLEMENTATION [14]. ALL RUNS WERE PERFORMED ON THE SAME TEST PLATFORM DESCRIBED UNDER EXPERIMENTAL SETUP. ENTRIES LABELED "N/A" DENOTE THE CASES WHERE THE SERIAL LOUVAIN IMPLEMENTATION CRASHED.

Input graph	Output modularity		Best run-time (in sec)		Heuristic reported (#threads)
	our imp.	serial Louvain	our imp.	serial Louvain	
CNR	0.912618	**0.912784**	0.569	4.210	baseline+Color (16)
coPapersDBLP	**0.858328**	0.848702	2.429	7.747	baseline+VF+Color (32)
Channel	**0.935286**	0.849672	24.471	28.279	baseline+Color (16)
Europe-osm	**0.998853**	n/a	66.453	n/a	baseline+VF+Color (32)
MG1	**0.968725**	0.968671	16.334	120.389	baseline+Color (32)
uk-2002	0.989363	**0.989700**	37.238	305.837	baseline+VF (32)
MG2	**0.998426**	**0.998426**	214.876	1099.061	baseline (32)

vertices to migrate to any one of the neighboring communities with even probability delaying convergence under a parallel setting. From the Table III, we can also note that the best run-times are mostly observed using either the baseline+Color or baseline+VF+Color.

VII. RELATED WORK

The literature on serial community detection is extensive and cannot be possibly reviewed within this paper. Therefore, we focus on some of the seminal works in the field and the recent developments in parallelization. Although the notion of community detection is not new, the field took a significant shape with the introduction of the modularity measure to quantify the quality of community outputs by Newman and Girvan in 2004 [2]. Newman's pioneering works on discovering community structure from networks also included developing both divisive [2], [15] and agglomerative [16] clustering methods. The divisive method use the edge betweenness centrality index to detect bridges between communities but due to the underlying computation involved, it is also very slow ($O(n^3)$ for sparse inputs), limiting its scalability to sparse networks with tens of thousands of vertices. The other class of algorithms use an agglomerative clustering approach where at any stage a greedy merging is performed between any two communities that provide the maximum modularity gain. This technique was originally introduced by the classical Clauset-Newman-Moore (CNM) algorithm [16] and since been adopted/tailored into many other methods (e.g., [17]). With an average time complexity of $O(n \log^2 n)$ this approach have shown better scaling to networks containing $\times 10^5 - 10^6$ nodes and $\times 10^6 - 10^7$ edges. The Louvain method [3] can also be thought of as a variant of this agglomerative strategy but with the key differences being that instead of carrying out the merging at a community-to-community level, the Louvain heuristic allows vertices to independently make decisions from within each community at every time step, and with a flexibility for those decisions to be undone at later iterations. Although input dependent, it has been shown that the Louvain approach is able to produce communities with better modularity scores than the other agglomerative strategies. On the other hand, the cluster hierarchies produced by agglomerative techniques tend to be more meaningful. For an extensive review on community detection methods and comparisons, please refer to [1], [18].

In the past few years, there have been several efforts in parallelizing modularity-based community detection. As part of the DIMACS10 clustering challenge, Riedy *et al.* presented a highly parallel agglomerative algorithm [19], [20] that is an extension to the CNM algorithm. The key difference is that their approach performs multiple pairwise community merges in parallel by treating the problem as one of an edge weighted matching. This strategy provides ample parallelism to the method and the authors present impressive performance and speedup figures on multiple data sets. As part of the same DIMACS10 challenge, Auer and Bisseling [21] present another way to achieve agglomerative clustering using GPUs. Their approach uses graph coarsening. This method also shows appreciable savings in time to solution. In a more recent study, Bhowmick and Srinivasan [22] present a shared memory parallel algorithm for the Louvain method. Their approach is to update the community structures on-the-fly from within each iteration as vertices are evaluated in parallel. This creates a need to introduce critical sections in parts which limits its scalability to small synthetic inputs ($\times 10^4$ vertices). The modularities reported also show variability across the processor spectrum. In another recent paper, Staudt and Meyerhenke [23] present an alternative approach that uses label propagation to parallelize the Louvain method. One of our future plans is to compare our method with other recently developed parallelization approaches such as the above.

VIII. CONCLUSION

In this paper, we introduced effective heuristics for parallelizing an important and widely used community detection method — the Louvain method. We attempted to address the dual objectives of maximizing concurrency, and retaining the quality with respect to serial implementations. To this end, we made two main contributions in this paper. First, we presented a detailed discussion of the heuristics for parallelization and provided analytical proofs of their effectiveness. Second, we empirically supported the observations with a set of carefully conducted experiments using seven real-world networks representing a diverse set of application domains. Compared to the serial Louvain implementation [14], our parallel implementation is able to produce community outputs with a higher modularity for most of the inputs tested, in comparable number of iterations, while providing real speedups of up to $8\times$ using 32 threads. In addition, our parallel implementation was able to exhibit weak scaling properties on up to 32 threads.

We believe that the mathematical discussion, heuristics, and experimental evidence provided in this paper will benefit a wide range of researchers dealing with increasingly larger data sets and continually weaker serial hardware performance. Our future work include: i) extending the experiments to larger-scale inputs with billions of edges; ii) a comprehensive comparison of communities produced by the serial and different parallel implementations by delineating differences by composition and not just modularity; iii) investigating the value of the vertex following heuristic in the context

of the serial Louvain algorithm and other modularity-based community detection algorithms; and iv) extension of our parallel algorithms to account for other modularity definitions (e.g., [5]) in order to overcome the well known resolution-limit issues of the standard modularity definition used in this paper.

ACKNOWLEDGMENT

The authors would like to thank Drs. Sanjukta Bhowmick and Assefaw Gebremedhin for initial discussions, and Dr. Emilie Hogan for reviewing a preliminary draft of the manuscript. The research was in part supported by DOE award DE-SC-0006516, NSF award IIS 0916463, and the Center for Adaptive Super Computing Software Multithreaded Architectures (CASS-MT) at the U.S. Department of Energy Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

REFERENCES

[1] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0370157309002841

[2] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.69.026113

[3] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008, 2008.

[4] DIMACS10, "The 10th DIMACS implementation challenge - graph partitioning and graph clustering," 00000. [Online]. Available: http://www.cc.gatech.edu/dimacs10/

[5] V. A. Traag, P. Van Dooren, and Y. Nesterov, "Narrow scope for resolution-limit-free community detection," *Physical Review E*, vol. 84, no. 1, p. 016114, 2011, 00039.

[6] D. Bader and J. McCloskey, "Modularity and graph algorithms," *SIAM AN10 Minisymposium on Analyzing Massive Real-World Graphs*, pp. 12–16, 2009, cited by 0004.

[7] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, 2011, cited by 0037.

[8] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 20, no. 2, p. 172âĂŞ188, 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4358966

[9] U. Catalyurek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, 2012, 00002.

[10] A. Azad, M. Halappanavar, S. Rajamanickam, E. Boman, A. Khan, and A. Pothen, "Multithreaded algorithms for maximum matching in bipartite graphs," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, 2012, pp. 860–872, 00000.

[11] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering: 10th DIMACS implementation challenge workshop," *Contemporary Mathematics*, vol. 588, Feb. 2012, 00000.

[12] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011, 00457.

[13] C. Wu, A. Kalyanaraman, and W. R. Cannon, "pGraph: efficient parallel construction of large-scale protein sequence homology graphs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 10, pp. 1923–1933, 2012, 00008.

[14] Louvain, "findcommunities," cited by 0000. [Online]. Available: https://sites.google.com/site/findcommunities/

[15] M. E. J. Newman, "Analysis of weighted networks," *Phys. Rev. E*, vol. 70, no. 5, p. 056131, Nov. 2004.

[16] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, no. 6, p. 066111, Dec. 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.70.066111

[17] K. Wakita and T. Tsurumi, "Finding community structure in mega-scale social networks:[extended abstract]." ACM, 2007, pp. 1275–1276, 00161.

[18] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, pp. 167–256, 2003.

[19] J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks." IEEE, 2012, pp. 1619–1628, 00005.

[20] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Parallel Processing and Applied Mathematics*. Springer, 2012, pp. 286–296, 00011.

[21] B. F. Auer and R. H. Bisseling, "Graph coarsening and clustering on the GPU." 2012, p. 223, 00006.

[22] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method for modularity maximization," in *Dynamics On and Of Complex Networks, Volume 2*. Springer, 2013, pp. 111–124, 00000.

[23] C. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," *Parallel Processing (ICPP), 2013 42nd International Conference on*, pp. 180–189, Oct. 2013.