Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 14-010

Multi-Threaded Modularity Based Graph Clustering using the Multilevel Paradigm

Dominique LaSalle and George Karypis

May 05, 2014

Multi-Threaded Modularity Based Graph Clustering using the Multilevel Paradigm

Dominique LaSalle and George Karypis
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota 55455, USA
{lasalle,karypis}@cs.umn.edu

Abstract—**Graphs are an important tool for modeling data in many diverse domains. Recent increases in sensor technology and deployment, the adoption of online services, and the scale of VLSI circuits has caused the size of these graphs to skyrocket. Finding clusters of highly connected vertices within these graphs is a critical part of their analysis.**

In this paper we apply the multilevel paradigm to the modularity graph clustering problem. We improve upon the state of the art by introducing new efficient methods for coarsening graphs, creating initial clusterings, and performing local refinement on the resulting clusterings. We establish that for a graph with n vertices and m edges, these algorithms have an $O(m+n)$ runtime complexity and an $O(m+n)$ space complexity, and show that in practice they are extremely fast. We present shared-memory parallel formulations of these algorithms to take full advantage of modern architectures. Finally, we present the product of this research, the clustering tool *Nerstrand*[1]. In serial mode, *Nerstrand* runs in a fraction of the time of current methods and produces results of equal quality. When run in parallel mode, *Nerstrand* exhibits significant speedup with less than one percent degradation of clustering quality. *Nerstrand* works well on large graphs, clustering a graph with over 105 million vertices and 3.3 billion edges in 90 seconds.

I. Introduction

Graphs are an important tool for representing data in many diverse domains. Graph clustering is a technique for analyzing the structure of a graph by identifying groups of highly connected vertices. Discovering this structure is an important task in social network, biological network, and web analysis. In recent years, the scale of these graphs has increased to millions of vertices and billions of edges, making this discovery increasingly difficult and costly.

Modularity [1] is one of the most widely used metrics for determining the quality of non-overlapping graph clusterings, especially in the network analysis community. The problem of finding a clustering with maximal modularity is NP-Complete [2]. As a result many polynomial time heuristic algorithms have been developed [3], [4], [5], [6], [7], [8]. Among these algorithms, approaches resembling the multilevel paradigm as used in graph partitioning have been shown to produce high quality clustering solutions and scale to large graphs [9], [10], [11].

However, most of these approaches adhere closely to the agglomerative method of merging pairs of clusters iteratively. This can lead to skewed cluster sizes as well as require excessive amounts of computation time. While methods for prioritizing cluster merges have been proposed to reduce skewed cluster sizes, these approaches are inherently serial. The use of post-clustering refinement has not been present in most of these approaches, and there is currently no parallel formulation for modularity refinement.

In this paper we present multilevel algorithms for generating high quality modularity-based graph clusterings. The contributions of our work are:

- A method for efficiently contracting a graph for the modularity objective.
- An robust method for generating clusterings of a contracted graph.
- A modified version of k-way boundary refinement for the modularity objective.
- Shared-memory parallel formulations of these algorithms.

We show that for a graph with n vertices and m edges, these algorithms in have an $O(m+n)$ time and $O(m+n)$ space complexities. To validate our contributions, we compare our implementation of these algorithms, *Nerstrand*, against the serial clustering tool *Louvain* [9] and the parallel clustering tool *community-el* [11], and show that *Nerstrand* produces clusterings of equal or greater modularity and is 2.7–44.9 times faster. We also compare the quality of clusterings generated by the serial version of *Nerstrand* against the results of the 10th DIMACS Implementation Challenge [12] on graph partitioning and graph clustering. The modularity of clusterings produced by *Nerstrand* are equal to or within only a few percentage points of the best clusterings reported in the competition while requiring several orders of magnitude less time. The parallel version of *Nerstrand* is scalable and extremely fast, clustering a graph with over 105 million vertices and 3.3 billion edges in 90 seconds using 16 cores.

This paper is organized into the following sections. In Section II we define the notation used throughout this paper. In Section III we give an overview of current graph clustering methods for maximizing modularity. In Section IV we give an overview of the multilevel paradigm and its use in the graph partitioning problem and more recently in the graph clustering problem. The descriptions of the serial algorithms we developed are presented in Section V, and the descriptions of their parallel counter parts are presented in Section VI. In

[1]The *Nerstrand* software is available at http://cs.umn.edu/~lasalle/nerstrand

Section VII we describe the conditions of our experiments. This is followed by the results of our experiments in Sections VIII and IX, in which we evaluate the quality and speed of the presented algorithms. Finally in Section X, we review the findings of this paper.

II. Definitions & Notation

A simple undirected *graph* $G = (V, E)$ consists of a set of vertices V and a set of edges E, where each edge $e = \{v, u\}$ is composed of an unordered a pair of vertices (i.e., $v, u \in V$). The number of vertices is denoted by the scalar $n = |V|$, and the number of edges is denoted similarly as $m = |E|$. Each edge $e \in E$ can have a positive weight associated with it that is denoted by $\theta(e)$. If there are no weights associated with the edges, then their weights are assumed to be one.

Given a vertex $v \in V$, its set of adjacent vertices (connected by an edge) is denoted by $\Gamma(v)$ and is referred to as the *neighborhood* of v. For an unweighted graph, $d(v)$ denotes the number of edges incident to v (e.g., $d(v) = |\Gamma(v)|$), and for the case of weighted edges, $d(v)$ denotes the total weight of its incident edges (e.g., $d(v) = \sum_{u \in \Gamma(v)} \theta(\{v, u\})$).

A *clustering* C of G is described by the division of V into k non-empty and disjoint subsets $C = \{C_1, C_2, \ldots, C_k\}$, which are referred to as *clusters*. The sum of vertex degrees within a cluster is denoted as $d(C_i)$ (i.e., $d(C_i) = \sum_{v \in C_i} d(v)$). The internal degree $d_{int}(C_i)$ of a cluster C_i is the number of edges (or sum of the edge weight) that connect vertices in C_i to other vertices within C_i. The external degree $d_{ext}(C_i)$ of a cluster C_i is the number of edges (or sum of the edge weight) that connect vertices in C_i to vertices in other clusters. The neighborhood of a cluster V_i, that is all clusters connected to C_i by at least one edge, is denoted by $\Gamma(C_i)$. The number of edges connecting the cluster C_i to C_j is denoted as $d_{C_j}(C_i)$. Since G is an undirected graph, $d_{C_j}(C_i) = d_{C_i}(C_j)$. Similarly, the number of edges (or total edge weight) connecting a vertex v to the cluster C_i is denoted as $d_{C_i}(v)$ (i.e., $d_{C_i}(v) = \sum_{u \in C_i \cap \Gamma(v)} \theta(\{v, u\})$). To aid in the discussion of moving vertices between clusters, we will denote the cluster C_i with the vertex v removed, as $C_i - \{v\}$, and the cluster C_j with the vertex v added as $C_j + \{v\}$.

The metric of graph *modularity*, and the focus of this paper, was introduced by Newman and Girvan [1], and has become ubiquitous in recent graph clustering/community detection literature. Modularity measures the difference between the expected number of intra-cluster edges and the actual number of intra-cluster edges. Denoted by Q, the modularity of a clustering C is expressed as

$$Q = \frac{1}{d(V)} \left(\sum_{C_i \in C} d_{int}(C_i) - \frac{d(C_i)^2}{d(V)} \right), \qquad (1)$$

where $d(V)$ is the total degree of the entire graph (i.e., $d(V) = \sum_{v \in V} d(v)$). From this, we can see the modularity Q_{C_i} contributed by cluster C_i is

$$Q_{C_i} = \frac{1}{d(V)} \left(d_{int}(C_i) - \frac{d(C_i)^2}{d(V)} \right). \qquad (2)$$

The value of Q ranges from -0.5, where are of the edges in the graph are inter-cluster edges, and approaches 1.0 if all edges in the graph are intra-cluster edges and there is a large number of clusters. Note that this metric does not use the number of vertices within a cluster, but rather only the edges. Subsequently, vertices of degree zero, can arbitrarily be placed in any cluster without changing the modularity.

III. Modularity Based Graph Clustering

Since its introduction a decade ago by Newman [1], a large number of approaches for maximizing the modularity of a clustering have been developed. Fortunato [13] provides an overview of modularity and methods for its maximization.

The majority of approaches fall into the category of agglomerative clustering. In agglomerative clustering, each vertex is placed in its own cluster, and pairs of clusters are iteratively merged together if it increases the modularity of the clustering. When there exists no pair of clusters whose merging would result in an increase in modularity, the process stops, and the clustering is returned.

The greedy agglomerative method introduced by Clauset et al. [4], is the most well-known of the these approaches, due to its ability to find good clusterings in relatively little time. Its low runtime is the result of exploiting the sparse structure of the graph to limit the number of merges it needs to consider and the number of updates that it needs to perform during agglomeration. The quality of the clusterings it finds is the result of recording the modularity after each merge, and continuing to perform cluster merges until there is only a single cluster, and then reverting to the state with the maximum modularity. The structure used to maintain this state information is a binary tree in which each node represents a cluster, and the children of a node are the clusters which were merged to form the node. They established an upper bound on the complexity of this algorithm of $O(mh \log n)$, where h is the height of the tree recording cluster merges. If this tree is fairly balanced, h will be close to $\log n$.

It was noted that this algorithm tends to discover several super-clusters, composed of most of the vertices in the graph. Wakita and Tsurumi [14] showed that these super clusters are the result of one or a few large clusters successively merging with small clusters, causing h to approach n, which results in a running time near $O(mn)$. They also showed that the creation of these super-clusters can be of detriment to the modularity of the clustering. They addressed this by presenting an algorithm that favors merging clusters of similar size, which helps to prevent this unbalanced merging.

The Louvain method [9] finds a set of cluster merges through an iterative process. It does this by initializing every vertex to its own cluster as is done in agglomerative methods, and then for each vertex, checks to see if moving it to a different cluster will improve modularity. It moves vertices this way in passes, until a pass results in no moves being made. Then, a new graph is generated where each vertex is a cluster of vertices from the previous graph. This process is repeated until a graph is generated in which no vertices change clusters. This is currently one of the fastest modularity based clustering methods available [15].

There is a small number of parallel algorithms for modularity based graph clustering. Reidy et al. [11] generate new graphs similar to the Louvain method. However, here instead of moving vertices, clusters are merged by collapsing a maximal matching of the clusters. Parallelism is extracted by calculating the desirability to collapse each edge independently, and then a multi-pass method is used to find the maximal cluster matching. Fagginger Auer and Bisseling [16] present a similar approach using maximal matchings on GPU architectures, with extensions to matching in order to increase the rate of cluster merging. Both of these use a fine grain approach to parallelism, and are similar to the coarsening phase of the multilevel paradigm discussed in the next section.

IV. The Multilevel Paradigm

Since their introduction over 20 years ago [17], multilevel approaches for graph partitioning have become the standard for developing high-quality and computationally efficient methods [18], [19], [20], [21], [22]. These algorithms solve the underlying optimization problem using a methodology that follows a *simplify & conquer* approach, initially used by multi-grid methods for solving systems of partial differential equations.

The multilevel paradigm is composed of three distinct phases: *coarsening*, *initial clustering*, and *uncoarsening*. In the coarsening phase, the original graph G_0 is used to generate a series of increasingly *coarser* (smaller) graphs, $G_1, \ldots, G_s$. In the initial clustering phase, a solution C_s of the much smaller graph G_s is generated using a direct clustering algorithm (i.e., a non-multilevel clustering algorithm). Finally, in the uncoarsening phase, the initial clustering is used to derive clusterings of the successive *finer* (larger) graphs. This is done by first *projecting* the clustering of G_{i+1} to G_i, followed by cluster refinement whose goal is to increase the modularity of the clustering by moving vertices between the clusters. Since the successive finer graphs contain more degrees of freedom, such refinement is often feasible and leads to significant modularity improvements.

Noack and Rotta [10] developed a method for modularity based graph clustering that uses the multilevel paradigm. Instead of collapsing independent sets of vertices as in graph partitioning, they use agglomerative clustering to iteratively determine groups of vertices to collapse together. To avoid the uneven merging of clusters, they prioritize cluster merges based on $\Delta Q/\sqrt{d(C_i)d(C_j)}$, where ΔQ is the gain in modularity from merging clusters C_i and C_j. The state of the clustering is intermittently instantiated as a graph to provide several levels on which refinement can be performed. Their refinement visits each vertex and considers it for moving between clusters.

Djidjev and Onus [23] showed that the multilevel algorithms of [18] for graph partitioning can be used directly to find two-way clusterings with high modularity by using a modularity derived input graph.

V. Serial Clustering Methods

The algorithms that we developed for *Nerstrand* follow the multilevel paradigm closely as it is used for the graph partitioning problem. A high-level overview of how these algorithms fit together is as follows:

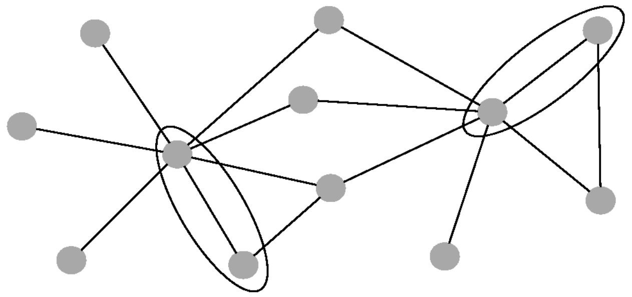

Fig. 1. A small maximal matching.

1) A graph $G_0 = (V, E)$ is given as *input*.
2) *Coarsening*: A series of increasingly coarser graphs is generated: $G_1, \ldots, G_s$.
3) *Initial Clustering*: A clustering $C = \{C_1, \ldots, C_k\}$ is created by assigning each vertex in G_s to a cluster.
4) *Uncoarsening*: The clustering C is projected through the series of coarse graphs, $G_s \rightarrow \cdots \rightarrow G_0$, while being improved for each graph.
5) A vector P mapping each vertex to a cluster ID, is returned as *output*.

We introduce aggregation schemes to address the issue of coarsening graphs with power-law degree distributions in Section V-A. We introduce a method for effectively generating initial clusterings of a coarsened graph in Section V-B. We present a formulation of refinement for maximizing modularity in Section V-C. We give a complexity analysis of these algorithms in Section V-D, showing that they run in $O(m+n)$ time and $O(m+n)$ space.

A. Coarsening

In coarsening, vertices in G_i are *aggregated* together to form the vertices of G_{i+1}. We explored three different aggregation schemes: *matching* (MAT), *matching with secondary two-hop matching* (M2M), and *first choice grouping* (FCG). For all three aggregation schemes, we attempt to merge each vertex, which helps to prevent the skewed cluster sizes present in greedy agglomerative methods. These three schemes choose vertices to aggregate together by selecting the vertex u to aggregate v with that maximizes the function $Q_{merge}(v, u)$. This is the change in modularity that would result if v and u were clusters and were merged to form a single cluster. The change in modularity by merging v and u is

$$Q_{merge}(v, u) = Q_{\{v,u\}} - (Q_{\{v\}} + Q_{\{u\}}). \qquad (3)$$

1) Matching: Our standard matching algorithm (MAT), visits each vertex v in random order and matches v with the unmatched neighbor $u \in \Gamma(v)$ for which equation (3) is maximized. If v has no unmatched neighbors, or equation (3) is below zero, v is matched with itself (aggregated by itself).

Standard matching works well on graphs with near uniform degree distribution as matchings tend to be very large. However, power-law degree distributions prevent large matchings, causing the coarser graph to be of similar size of the fine graph. An example of this is shown in Figure 1, where a maximal matching of a graph with twelve vertices has a size of two.

2) Two-Hop Matching: To address these matchings of small size in MAT, we developed a variation of matching that uses secondary two-hop matching (M2M). In this scheme when all of the neighbors of v are matched, v matches with an unmatched neighbor of one of its neighbors. That is, v matches with a vertex $w \in \Gamma(u)$, where $u \in \Gamma(v)$. There is no prioritization in finding w and instead the first unmatched w is used. This is due to computational cost that would be associated with performing a complete scan of all two-hop neighbors. To ensure we do not decrease the quality of the matching, we limit two hop matching to only vertices with low degree.

3) First Choice Grouping: The third option we explored for aggregating power-law graphs was to allow more than two vertices to be grouped together into a coarse vertex. The first choice grouping (FCG) scheme is based on the FirstChoice aggregation scheme originally used for contracting hypergraphs [24] and later applied to contracting simple graphs for the graph partitioning problem [25]. Our formulation for this paper differs from these earlier methods in that we not only consider the weight of the edge, but the current state of vertex groupings and the associated modularity gain.

When searching for a vertex or vertices to aggregate the vertex v with, all of the neighbors $u \in \Gamma(v)$ are considered regardless of whether they have been matched/grouped already. If u is ungrouped, then its priority for grouping is determined using equation (3). If u belongs to the group g, then the priority for adding v to that group is determined similarly, except the edges from g to v need to be summed, and $d(g)$ needs to be tracked.

B. Initial Clustering

Once coarsening is finished, we are left with the coarsest graph G_s, and need to create the clustering $C = \{C_1, C_2, \ldots, C_k\}$, and a vector mapping vertices to clusters P, where $|P| = |V|$ and $1 \leq P(v) \leq k$, $\forall v \in V$. We call this process *initial clustering*. Initial clustering is done with a direct clustering scheme, that is, a non-multilevel scheme that operates directly on G_s.

In G_s, each vertex is the result of collapsing together clusters of fine vertices during coarsening. For this reason, we can use a relatively simple initial clustering scheme. Our initial clustering scheme works by initializing each vertex to its own cluster as in agglomerative clustering, we then apply refinement as described in Section V-C2. This is similar to a single level of the Louvain [9].

C. Uncoarsening

In the uncoarsening phase, we take the clustering of the coarsest graph, G_s, and use it as an estimate for a good clustering of the finer G_{s-1}. We then improve it for G_{s-1} finding a local maxima of modularity. This is repeated until the clustering is applied to, and improved for G_0. The process of applying the clustering of G_i to G_{i-1} is referred to as *projection*. The process of improving the clustering for G_{i-1} is referred to as *refinement*.

1) Projection: Projection in *Nerstrand* is done by propagating cluster information from the coarse vertices in G_{i+1} to the fine vertices in G_i. By keeping track of what fine vertices compose a coarse vertex, we can project a clustering of G_{i+1} to G_i, by assigning each fine vertex in G_i to the same cluster that its coarse vertex is assigned. Since we keep track of collapsed edge weight for each coarse vertex, and use them in computing cluster degrees, the modularity of the clustering does not change in projection.

2) Refinement: We developed two modularity based refinement methods: *Random Boundary Refinement*, and *Greedy Boundary Refinement*. These two methods differ only in the order in which they consider vertices for moving. Both methods visit only vertices that are connected via an edge to one or more vertices which reside in different clusters. These vertices are referred to as *boundary* vertices. Similarly, when considering moving a vertex, we only evaluate the gain associated with moving it to a cluster to which it is connected.

It is possible that moving a vertex to a cluster to which is not connected or moving a vertex that is not a boundary vertex could result in a positive gain in modularity. For this to occur, when moving the vertex $v \in C_i$ to the cluster C_j to which it has no connection, the difference in the degree of C_i and the degree C_j must make up a larger fraction of the total edge weight in the graph than the fraction of v's edge weight that connects to C_i:

$$\frac{d(C_i - \{v\}) + d(C_j)}{d(V)} > \frac{d_{C_i}(v)}{d(v)}.$$

We observed that when considering all vertices for movement to all clusters resulted in only a 0.06% gain in modularity, while taking over 16 times as long. Furthermore, Brandes et al. [2] showed that a clustering with maximum modularity does not include non-contiguous clusters.

The gain by moving a vertex from cluster V_i to cluster V_j is given by the combined change in the cluster modularities:

$$\Delta Q = (Q_{C_i - \{v\}} + Q_{C_j + \{v\}}) - (Q_{C_i} + Q_{C_j}).$$

Note that if it leads to a positive gain in modularity, clusters can be completely emptied and removed during refinement.

If at least one vertex was moved while visiting all of the boundary vertices, another pass is performed. Refinement stops when no vertices are moved in a pass, or when a maximum number of passes has been made.

Random Boundary Refinement visits the boundary vertices in random order. This has two advantages. The first is that we can visit all of the boundary vertices in linear time. The second is that it is stochastic, and we can perform it multiple times using the same input clustering with different random seeds to explore the solution space.

Greedy Boundary Refinement inserts the boundary vertices into a priority queue. Each vertex is then extracted from this priority and considered for moving to a different cluster. As the state of the clustering changes, the priority of the vertices remaining in the priority queue is updated. This ensures that we continually make the best available move for the current clustering state.

To accurately prioritize vertices for movement between clusters based on modularity gain, we would need to use:

$$\Delta Q = (Q_{C_i - \{v\}} - Q_{C_i}) + \arg\max_j (Q_{C_j + \{v\}} - Q_{C_j}). \quad (4)$$

This however, is an expensive priority to maintain as the $\arg\max$ part of the equation will change each time a vertex is moved to or from one of the clusters to which v is connected.

We decided instead to use a heuristic for the priority. This heuristic uses the modularity gain associated with removing the vertex from its current cluster only (the left side of equation (4)). Using this priority, boundary vertices are inserted into a priority queue. Vertices are then extracted from the priority queue and the modularity gains associated with moving the front vertex are evaluated fully. In our experiments we did not observe an increase in the modularity of clusterings if we kept it up to date.

D. Complexity Analysis

The overall complexity for the serial algorithms in *Nerstrand* is the sum of its three phases. Adding the complexity of these phases: coarsening ($O(m+n)$, Section V-D2), initial clustering ($O(m+n)$, Section V-D3), and uncoarsening ($O(m+n)$ for RBR and $O(m \log n)$ for GBR, Section V-D4), we get an overall computational complexity of $O(m+n)$ (and $O(m \log n)$ if GBR is used). The space complexity is determined by the combined size of the generated graphs, which we show to be $O(m+n)$ in Section V-D1.

1) Upper Bound on Total Vertices and Edges: The total number of vertices and edges in the entire series of graphs $G_0, \ldots, G_s$, determines the input size for many of the algorithms in *Nerstrand*. If only a single edge is collapsed between successive graphs such that $|n_{i+1}| = |n_i| - 1$ and $|m_{i+1}| = |m_i| - 1$, the total number of vertices and edges processed would be n^2 and m^2 respectively giving a memory and computational complexity of at least $O(m^2 + n^2)$. We address this issue by stopping coarsening when the rate of contraction slows beyond $|G_i| > \alpha |G_{i-1}|$ where $0 < \alpha < 1.0$. Here $|G|$ represents the size of the graph, this can be in terms of the number of vertices, the number of edges, or a combination of the two. The total number of vertices and edges processed can then be represented as the sum of a geometric series:

$$\sum_{i=0}^{s} |G_i| = \sum_{i=0}^{s} |G_0| \alpha^i = |G_0| \frac{1 - \alpha^{s+1}}{1 - \alpha}, \quad (5)$$

and since a graph must contain at least one vertex (and for our purposes at least one edge) we can place on upper bound on s of $\log_\alpha(1/|G_0|)$. Plugging this in for s in equation (5) we get

$$|G_0| \frac{1 - \alpha^{\log_\alpha(\frac{1}{|G_0|})\alpha}}{1 - \alpha} = |G_0| \frac{1 - \frac{\alpha}{|G_0|}}{1 - \alpha} < \frac{|G_0|}{1 - \alpha}.$$

Since α is a constant, we can see then that the total number of vertices is $O(n)$ and the total number of edges is $O(m)$. Our choice of α not only changes the constants involved in these complexities, but also the size of G_s, which affects the quality of the clustering and the amount of computation required during initial clustering.

2) Coarsening Complexity: In the standard matching aggregation scheme (MAT), each vertex v chooses the unmatched neighbor that maximizes equation (3). This requires each vertex to scan through all of its edges, which makes this an $O(m+n)$ operation.

In the two-hop matching aggregation scheme (M2M), each vertex v chooses one of its unmatched neighbors to match with, or one of its neighbor's unmatched neighbors. When unrestricted, in a worse case scenario this would result in the scanning of the edges of all of v's neighbors, $d(\Gamma(v))$, which would make this an $O(m^2)$ operation. To keep the complexity to $O(m+n)$, we limit the total number of neighbor's edges scanned by v to a constant number (we use 32), before it is matched with itself.

In the first choice grouping aggregation scheme (FCG), each vertex v chooses one of its neighbors with which to match. The degree of groupings are updated incrementally as they are formed in $O(1)$ time, which allows determining the degree of a grouping g in $O(1)$ time. As v scans through its edges to determine with whom to match, it sums up the weight of edges connected to grouping using a hash table, which takes $O(1)$ time per edge. This allows us to look up $d_v(g)$ in $O(1)$. As a result, FCG can be done in $O(m+n)$.

To construct G_{i+1} based on the aggregation of G_i, we iterate over the set of vertices V_i in G_i. When we encounter a vertex $v \in V_i$ that is matched with a vertex $u \in V_i$ with a lower label (or with itself), we construct the new vertex $c \in V_{i+1}$. We merge the adjacency lists of v and u via a hash table using the corresponding coarse vertex numbers as keys. This allows us to combine edges to a vertex $w \in \Gamma(v), \in \Gamma(u)$ as well as edges to vertices x and y that have been aggregated together. This translates to operating on each vertex in the graph and inserting each edge into a hash table which is an $O(1)$ operation, which also gives us a complexity of $O(m+n)$ for contracting a graph with n vertices and m edges. Thus, coarsening G_i to G_{i+1} requires $O(m_i + n_i)$ time, and storing G_{i+1} requires $O(m_{i+1} + n_{i+1})$ space. Using the result from Section V-D1, we can then say that the coarsening phase takes $O(m+n)$ time.

3) Initial Clustering Complexity: In order to analyze the complexity in the context of initial clustering, let $n_s = |V_s|$ and $m_s = |E_s|$ represent the number of vertices and number of edges in G_s, respectively. Assigning clusters to all vertices requires $O(n_s)$ time and $O(n_s)$ space, since this only requires giving each vertex a label. Performing a pass of Random Boundary Refinement on the n_s clusters then takes $O(n_s + m_s)$ time as described in Section V-D4. A constant number of clusterings are created, and since at initial clustering $n_s \le n$ and $m_s \le m$, initial clustering has a computational complexity of $O(n+m)$.

4) Uncoarsening Complexity: As projection is a simple lookup in two arrays for each vertex in the fine graph G_i, projection is an $O(n_i)$ operation per graph. Since we know that there are $O(n)$ vertices total in all of the graphs of the multilevel hierarchy, we know that the total complexity of projection is $O(n)$.

In Random Boundary Refinement, the list of boundary vertices can be permuted in $O(n_i)$ time. Each vertex v is visited once per pass, and at most $d(v)$ edges will be inspected when deciding to move v, and at most $d(v)$ clusters will need to be updated if v is moved. So in the worse case we will need to visit n_i boundary vertices, and we may need to inspect up to m_i edges, and if every vertex is moved then m_i cluster updates will need to be performed. This gives us a complexity of $O(m_i + n_i)$ per pass. By limiting the number of passes that can be performed to a constant number (we found eight to work well), we can see that Random Boundary Refinement takes at most $O(m + n)$ time.

Greedy Boundary Refinement performs the same operations as Random Boundary Refinement with the addition of inserting, updating, and extracting vertices from the priority queue, which dominates the runtime. The priority queue contains up to n_i vertices and up to m_i updates can be performed upon it. Which means per graph, refinement takes $O(m_i \log n_i)$ time using a binary heap implementation [26]. And then for all of the graphs in the multilevel hierarchy using the result from Section V-D1 we have:

$$O\left(\sum_{i=0}^{s} m_i \log n_i\right) \leq O\left(\left(\sum_{i=0}^{s} m_i\right) \log\left(\sum_{i=0}^{s} n_i\right)\right).$$

We previously established that $\sum_{i=0}^{s} m_i = O(m)$ and $\sum_{i=0}^{s} n_i = O(n)$, so we know that the total complexity of refinement is $O(m \log n)$. This however is quite pessimistic for the objective of modularity, as it tends to favor large clusters as shown by Fortunato and Barthelemy [27], and good clusterings tend to have a set of *core* vertices on the interior of clusters (not part of boundaries) as shown by Ovelgönne and Geyer-Shulz [28].

VI. Parallel Clustering Methods

In order to allow *Nerstrand* to take advantage of modern compute architectures, we developed shared memory parallel versions of the previously outlined algorithms. We developed a method for assigning vertices to threads in a manner that balances the number of edges a thread is responsible for while partially preserving data locality resulting from the initial ordering. For parallelizing the coarsening phase, we introduce a method for contracting groups of vertices together in an unprotected fashion and resolving broken groupings. Finally, we introduce a method for performing k-way modularity refinement in parallel.

Our general approach to parallelization follows a coarse-grained model, where threads allocate their own memory and synchronization points are minimized [29]. Each thread manages its own subset of vertices of the original graph. We use the CSR sparse matrix data structure for storing the graph. Each thread allocates its own CSR structure to store its vertices and incident edges. Each thread is responsible for performing the computation associated with its vertices and edges.

A. Graph Distribution

The distribution of the vertices of the graph among the threads can have a significant impact on performance. We can explicitly balance the number of vertices per thread by assigning n/p vertices to each thread, where p is the number of threads. However, this could result in a significant imbalance of edges if high degree vertices are assigned to the same thread. Similarly, if we explicitly balance the number of edges by assigning vertices to threads such that the sum of the degrees is roughly $2m/p$, this could result in significant imbalance of vertices if a thread is assigned only a few high degree vertices.

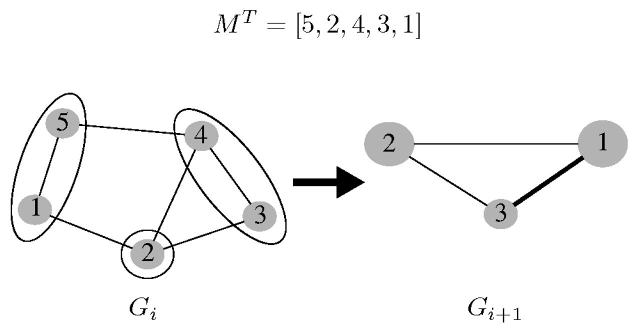

Fig. 2. The value of M when generating G_{i+1} from G_i.

Because the number of edges in a graph is often much greater than the number of vertices, we choose to explicitly balance the number of edges assigned to each thread and implicitly balance the number of vertices assigned to each thread. This is accomplished by re-ordering the vertices via a block-random permutation and then assigning a continuous chunk to each thread such that sum of the degrees is roughly $2m/p$. That is, the vertices are first blocked together (we use a block size of 1024) via their initial ordering, and then the blocks are re-ordered randomly. This block-random permutation ensures that with high probability the degree distribution of the vertices owned by each thread is similar, and thus the number of vertices assigned to each thread will also be similar.

We do this instead of re-ordering the vertices individually as data locality can have a significant impact on performance, and the input graph may have an ordering with strong locality. An ordering with strong locality is one in which vertices that are proximal to each other on the graph are also stored proximal to each other in memory. This allows for better utilization of processor cache and of memory bandwidth. By performing the permutation in blocks, we are able to preserve this locality within the blocks. Because multiple blocks will be assigned to each thread, even if a thread is assigned a block containing high degree vertices, the degree distribution among all of its assigned blocks will still likely reflect the degree distribution of the graph. As a result, the number of vertices assigned to it should not be significantly different from that of other threads.

B. Coarsening

For parallelizing aggregation, we update the data structures for recording vertex matchings/groupings without using locks or exclusive access patterns, allowing race conditions. We then fix the broken matchings caused by race conditions after attempting to match/group all vertices.

A matching vector M stores information about the other vertices that are part of a coarse vertex via symmetric matchings. So if the vertex v is matched with the vertex u, then $M(v) = u$, and $M(u) = v$. For a given vertex v, let $M(v) = u$, where u is the next vertex that has been matched with v. In the example given by Figure 2, two pairs of vertices are matched, and one vertex is matched with itself. Vertices 1 and 5 are matched, so $M(1) = 5$ and $M(5) = 1$, vertices 3 and 4 are matched, so $M(3) = 4$ and $M(4) = 3$, and vertex 2 is matched with itself, so $M(2) = 2$.

A broken matching is where $M(v) = u$, but $M(u) \neq v$, which can be caused if one thread is tries to match v with u, and another thread tries to match w with u and overwrites $M(u)$ with w. After threads finish matching their vertices, they re-iterate over them and for any vertex v for which $M(M(v)) \neq v$, the vertex is matched with itself, $M(v) = v$. This technique for performing parallel matching was first proposed by Catalyürek et al. [30], and can be directly applied to the MAT and M2M schemes.

This however, does not apply to aggregation schemes where more than two vertices can be aggregated together at once, as is the case with FCG. To address this, we developed a method for parallel grouping vertices in an unprotected fashion. First we generalize M from being a matching vector to that of a *grouping* vector, where aggregated vertices in M form a cycle of arbitrary length. If a grouping contains the vertices v, u, and w, then $M(v) = u$, $M(u) = w$, and $M(w) = v$.

At the start of the aggregation step, all vertices are initially matched with themselves, $M(v) = v$. Then, to add the vertex v to the vertex u's grouping, we set $M(v) = M(u)$, and $M(u) = v$. This means that a valid grouping vector M will contain only cycles. However, performing updates to this vector without synchronization allows for broken cycles. Because M is initialized to be all length one cycles, and every write to M is a valid vertex number, we know that every index in M is a valid vertex number, and thus a valid index in M. Then, for every vertex v, the linked list created by following the indices $M(v), M(M(v)), \ldots, M(u)$, must be non-terminating, so we know that v must either be part of a cycle, or part of a *tail* connected to a cycle. For architectures in which the writing of words is not atomic (i.e., two threads writing to the same location could result in valid vertex number being written), a simple validity check can be added.

In order to cleanup these tails, the following method is used which does not require synchronization. Each thread marks all of its vertices as not finalized. Then for each vertex v that a thread owns that is not marked as finalized, the indices in M are followed until a cycle is found. If v is part of that cycle, and v is the owner of the cycle (we use the lowest vertex number in the cycle as the owner), then all vertices in the cycle are marked as finalized. If v is not part that cycle, it is matched with itself. This leaves us with a valid M vector where every vertex is part of a cycle (including cycles of length one). To avoid creating large cycles, during aggregation the size of groups of vertices are tracked, and vertices are only allowed to join groups smaller than a maximum size (we use 1024).

C. Initial Clustering

For a moderate number of threads, the initial clustering stage lends itself well to parallelization, where each thread creates one or more of the initial clusterings, and a reduction operation is performed at the end to choose the best one. That is, each thread performs Random Boundary Refinement on the coarse graph with each vertex initialized to a cluster. The only concern for parallelization here is effectively using the cache hierarchy to reduce the total memory bandwidth required. For large numbers of threads and sufficiently large coarse graphs, the threads work cooperatively to create each initial clustering using the parallel formulation of refinement described below in Section VI-D.

D. Uncoarsening

Projection is also an inherently parallel process, as each thread can independently perform cluster projection on the vertices it owns. Conversely, refinement is an inherently serial process.

Because the gains associated with moving a vertex v from the cluster C_i to the cluster C_j depends on the degree information C_i and C_j, we cannot guarantee that moving vertices in parallel will result in a positive net gain in modularity. Having the owning thread lock the pair of clusters C_i and C_j before moving the vertex v would allow us to guarantee we only make positive gain moves, but this would greatly limit the amount of parallelism.

Instead, each thread then makes a private copy of the global clustering state. This private copy is updated by the thread as it moves the vertices that it owns. Because each thread is unaware of the moves being made by other threads, a move that it sees a positive gain move, may actually result in a loss of modularity.

After all threads have made their desired moves, the global clustering state is updated. Each thread then makes a pass over its selected set of moves, a *roll-back* pass, where it re-evaluates each of its moves in reverse order. If, with the updated cluster information, the move no longer results in a positive gain, the move is rolled back. Note that this does not guarantee that no negative gain moves will be made, as rolling back moves in parallel has the same issue as making the initial moves in parallel. To guarantee no modularity loss, the roll-back pass would need to be repeated until no moves were rolled back, and all remaining moves would have been determined positive gain moves based on up-to-date cluster degrees. We opted to use only a single pass to keep the cost of refinement down as we found it sufficient to prevent the majority of negative gain moves. After all of the threads have rolled back undesirable moves, the global clustering information is updated, and another iteration is started.

The clusters in which the neighbors of v reside affects how the internal and external cluster degrees are effected by moving the vertex v. When performing refinement serially, this is not an issue, as only one vertex moves at a time, and cluster degrees can be updated directly.

Consider the edge $\{v, u\}$ and the incident vertices $v \in C_i$ and $u \in C_j$. If the vertex v is moved to C_j and the vertex

u is moved to C_k concurrently, if the thread that owns v directly updates the cluster degrees, then $2\theta\{v,u\}$ to be added to $d_{int}(C_j)$, when the edge is actually between C_j and C_k. Note that if v and u are owned by the same thread, this is not an issue as v and u will not be moved concurrently.

To solve this problem, we developed a method for handling cluster degree updates that is order independent. Our new method of processing cluster degree updates splits the updates into two distinct parts: *move updates* made by the moving vertex v, and *neighbor updates* made by each neighbor of v. Neighbor updates can be classified as local, where the moving thread also owns the neighbor, and as remote, where the neighbor is owned by a different thread. Move updates and local neighbor updates are applied to the private copies of the cluster degrees as moves are made. Remote neighbor updates are applied afterwards as part of the global clustering state update.

For the move update, the thread that owns the moving vertex v updates its local cluster degrees. For updating the source cluster C_i's internal degree

$$\Delta d_{int}(C_i) = -d_{C_i}(v),$$

C_i's external degree

$$\Delta d_{ext}(C_i) = \frac{d_{ext}(v) - d_{C_i}(v)}{2},$$

the destination cluster C_j's internal degree

$$\Delta d_{int}(C_j) = d_{C_j}(v), \tag{6}$$

and C_j's external degree

$$\Delta d_{ext}(C_j) = \frac{d_{ext}(v) + d_{C_i}(v) - d_{C_j}(v)}{2}.$$

For the neighbor update, the thread that owns the adjacent vertex u to the moving vertex v performs updates associated with the edge $\{v,u\}$. The source cluster C_i's internal degree is changed by

$$\Delta d_{int}(C_i) = \begin{cases} -\theta(\{v,u\}) & \text{if } u \in C_i \\ 0 & \text{else} \end{cases},$$

and its external degree is changed by

$$\Delta d_{ext}(C_i) = \begin{cases} \theta(\{v,u\})/2 & \text{if } u \in C_i \\ -\theta(\{v,u\})/2 & \text{else} \end{cases}.$$

The destination cluster C_j's internal degree is changed by

$$\Delta d_{int}(C_j) = \begin{cases} \theta(\{v,u\}) & \text{if } u \in C_j \\ 0 & \text{else} \end{cases}, \tag{7}$$

and its external degree is changed by

$$\Delta d_{ext}(C_j) = \begin{cases} -\theta(\{v,u\})/2 & \text{if } u \in C_j \\ \theta(\{v,u\})/2 & \text{else} \end{cases}.$$

By splitting the updates like this, they can be applied independent of the order in which the vertices were moved.

Applying this to our previous example where the vertex v is moved to C_j and the vertex u is moved to C_k concurrently, these order independent updates result in the correct cluster degree changes. First, $\theta\{v,u\}$ would get added to $d_{int}(C_j)$ as part of the move update via equation (6), and then the neighbor update performed after u has moved to C_k then removes $\theta\{v,u\}$ from $d_{int}(C_j)$ via equation (7). This has the correct net effect of leaving $d_{int}(C_j)$ unchanged with respect to the edge $\{v,u\}$.

To visit vertices in order of their potential gain for performing Greedy Boundary Refinement in parallel, each thread maintains a priority queue containing the boundary vertices which it owns. This means vertices are not strictly visited in descending order of their priority across threads. As shown by our experiments in Section IX-A, this has minimal effect on the modularity. For Random Boundary Refinement, each thread visits the boundary vertices it owns in random order.

TABLE I
GRAPHS USED IN EXPERIMENTS

Graph	# Vertices	# Edges
cit-Patents[31]	3,774,768	16,518,947
soc-pokec[32]	1,632,803	22,301,964
soc-LiveJournal1[33]	4,846,609	42,851,237
europe.osm[12]	50,912,018	54,054,660
com-orkut[34]	3,072,441	117,185,083
uk-2002[12]	18,520,486	261,787,258
com-friendster[34]	65,608,366	1,806,067,135
uk-2007-05[12]	105,896,555	3,301,876,564

VII. EXPERIMENTAL METHODOLOGY

The experiments that follow were run on a HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz system with 256GB of memory. We used GCC 4.7 and the accompanying libgomp that conforms to the OpenMP 3.1 specification. Unless otherwise noted, all runs were repeated 25 times with different random seeds to get the geometric mean, minimum, or maximum time and modularity.

We used three different codes for the experiments. The serial and parallel algorithms presented in the previous sections are implemented in *Nerstrand*, available at http://cs.umn.edu/~lasalle/nerstrand. When run with a single thread, a separate set of functions implementing the serial algorithms are executed. For simplicity, we will refer to single threaded executions of *Nerstrand* as *s-Nerstrand*, and the multi-threaded executions as *mt-Nerstrand*.

We compare *s-Nerstrand* against what is currently the fastest [15] available method for modularity maximization on large graphs, *Louvain* [9]. We used version 0.2 which is available from https://sites.google.com/site/findcommunities/. We also compare *mt-Nerstrand* against the parallel clustering tool *community-el* [11] using version 0.7, available at http://www.cc.gatech.edu/~jriedy/community-detection/.

Table I shows the graphs primarily used for evaluation in Sections VIII and IX. Some of these are directed graphs, but for these experiments we created undirected versions to be compatible with the modularity objective. Where we present the average modularity or average time across these graphs, we use the geometric mean to calculate the average.

We also used graphs from the 10th DIMACS Implementation Challenge [12]. This challenge consisted of two parts, graph partitioning and graph clustering. Participants submitted algorithms and subsequent implementations to compete in

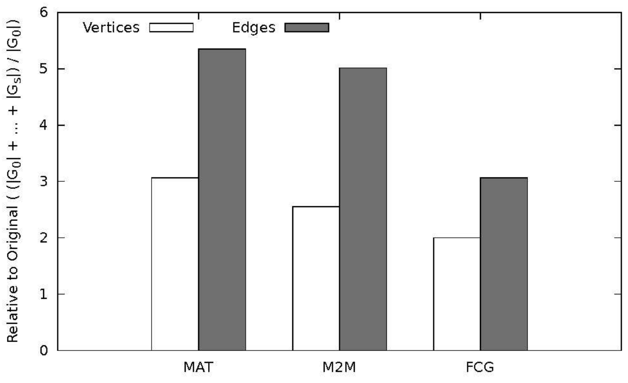

Fig. 3. Total vertices/edges of the coarsening scheme.

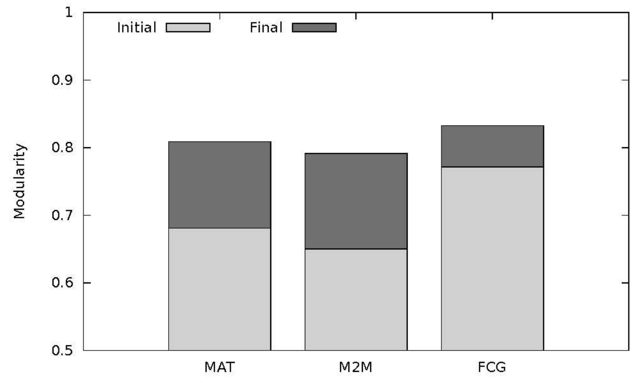

Fig. 5. Geometric mean modularity of each coarsening scheme.

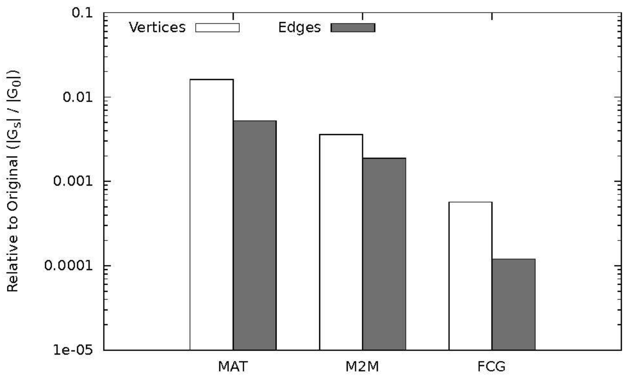

Fig. 4. Size of coarsest graph for each coarsening scheme.

several categories, one of which was modularity maximization. The results from this challenge are used for comparison in the next section.

VIII. Serial Results

In this section we present the results of our experiments designed to evaluate the different schemes for coarsening, initial clustering, and uncoarsening. For the evaluation of coarsening and initial clustering schemes, we used the graphs described by Table I, with the exception of *com-friendster* and *uk-2007-05*, which were excluded due to their size.

This is followed by the comparison of the best of these schemes as implemented in *s-Nerstrand* compared against the *Louvain* method. This comparison is in terms of clustering quality as well as runtime performance. All eight graphs from Table I were used for the comparison.

A. Aggregation Schemes

We evaluated the three coarsening schemes (MAT, M2M, and FCG) using several criteria. The first was rate of contraction. We measured this using the total number of vertices and edges found in G_0 through G_s, as this directly correlates to the amount of work done in the coarsening and uncoarsening phases. This is shown in Figure 3. The plain vertex matching scheme, MAT, did the worst, on average generating a total of 3.1 times as many vertices and 5.4 times as many edges as in the original graph. M2M did better, generating a total of 2.6 times as many vertices and 5.0 times as many edges as in the original graph. This improvement is the result of a more complete matching made possible by two-hop matches. Notice however that this primarily resulted in fewer vertices being generated, and only marginally decreased the number of edges generated. This is because when a two-hop match is made, edges are only combined, not collapsed. FCG did the best, generating only 1.7 times as many vertices and 2.8 times as many edges as in the original graph. This is because more than two vertices are aggregated together at a time, greatly reducing the number of vertices in coarser graphs, and increasing the number of edges that get combined. In addition to this, because we are targeting groups of highly connected vertices for collapsing, we contract a large number of edges with each coarse graph generated.

Next we evaluated the size of the final graph. That is, when $|G_s| > \alpha|G_{s-1}|$, as described in Section V-A. Figure 4 shows the number of vertices/edges in the coarsest graph divided by the number of vertices/edges in the original graph shown (y-axis is in log-scale). MAT again did the worst, contracting the final graph to 0.016 of its original vertices and 0.0052 of its original edges. M2M improved upon MAT, reducing the final graph to 0.0036 of its original vertices and 0.0019 of its original edges. This improvement can be attributed to the increased contraction rate of M2M. FCG did the best, reducing the final graph to 0.00021 of its original vertices and 0.000031 of its original edges. Notice that FCG significantly reduced the density of the final graph. This is because as FCG merges groups of vertices together, these groups are supposed to represent clusters, which by definition should have a large number of internal edges, and few external edges.

Our final evaluation criteria for the coarsening schemes was their effect on modularity. Figure 5 shows the modularity at after the initial clustering of the coarsest graph, as well as the final modularity of clustering refined and applied to the original graph. At the initial clustering phase, M2M did the worst, with an average modularity of 0.650, followed by MAT at 0.681. This difference in modularity between the two matching schemes can be attributed to the gain agnostic two-hop matches allowed by M2M. FCG did the best, with an average modularity 0.771. This is because where the two matching schemes will only choose the maximum gain matching from unmatched neighbors, FCG selects the maximum gain matching/grouping from among all neighbors.

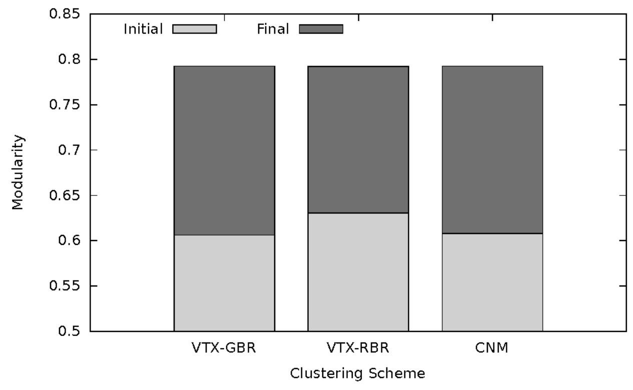

Fig. 6. The geometric mean modularity of initial clustering schemes.

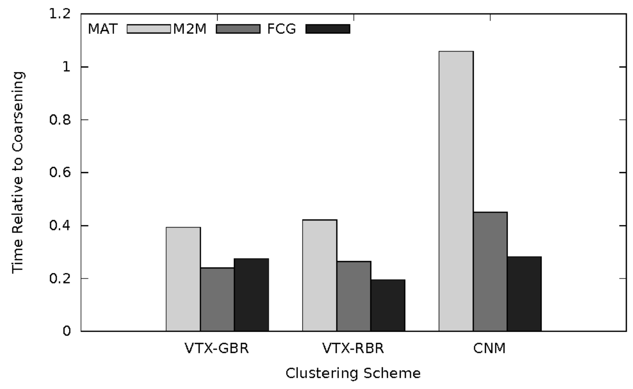

Fig. 7. The geometric mean runtime of initial clustering schemes.

After refinement, MAT and M2M were much closer, averaging 0.809 and 0.791 respectively. The reason for M2M closing the modularity gap, is that in refinement, many of the negative gain two-hop matches are undone. FCG still did the best, with an average of 0.832.

Due to the success of FCG in both reducing the size of the graph and in terms of modularity, we elected to use it as the coarsening scheme in *Nerstrand*.

B. Initial Clustering Schemes

A good initial clustering scheme will have a relatively stable runtime and solution quality over a variety of inputs. To better observe their robustness, we evaluated the initial clustering scheme in the context of all three coarsening schemes (MAT, M2M, and FCG).

We evaluated generating the initial clustering by initializing each vertex to its own cluster and refining it using both Greedy Boundary Refinement (VTX-GBR) as well as using Random Boundary Refinement to generate several clusterings and choosing the best (VTX-RBR). For these experiments we generated 16 different clusterings. We compared these two approaches to that of a modified version of the algorithm by Clauset et al. [4] that takes into account the weight of collapsed edges (CNM).

Figure 6 shows the quality of the initial clustering solutions both before and after refinement. The VTX-RBR method generated clusterings with the highest modularity at the end of initial clustering, 0.630. The VTX-GBR and CNM algorithms were similar in performance with modularities of 0.606 and 0.608 respectively. However, once refinement was run on these initial clusterings all three schemes had a final modularity of 0.792.

Figure 7 shows the amount of time spent in initial clustering and uncoarsening for each coarsening scheme relative to the amount of time spent in coarsening. The fastest overall initial clustering scheme was VTX-RBR, taking 27.9% of the time of coarsening. Only slightly slower, was VTX-GBR, taking 29.6% of the time of coarsening. VTX-RBR managed to be faster than VTX-GBR even though it made 16 clusterings, different complexities of vertex traversal: random permutation versus a priority queue. The CNM algorithm was the slowest, taking 51.3% of the time of coarsening.

Based on these findings, we selected VTX-RBR as the initial clustering scheme for use in *Nerstrand*.

C. Refinement Schemes

TABLE II
COMPARISON OF REFINEMENT SCHEMES.

Method	Mod. Improvement	Runtime (s)
RBR	0.01705	3.62906
GBR	0.01708	8.08687

The effect of the two different refinement schemes (RBR and GBR) on modularity as well as their runtimes are shown in Table II. Their modularity improvement was nearly identical, with GBR improving modularity only 0.15% more than RBR. Although the order in which vertices were visited during refinement appears to to not impact the modularity improvement, it does however affect the number of refinement passes required at each level. GBR on average made 1.07 passes before reaching a steady state, whereas RBR made an average of 2.10 passes before reaching a steady state. The cost of maintaining the priority queue caused GBR to be significantly slower, taking 2.23 times longer than RBR. This is a product of the $O(\log n)$ time required to insert, update, and remove vertices from the priority in GBR, as opposed to the randomly permuted list used in RBR in which vertices are only inserted in $O(1)$ time.

Given that both schemes improve the quality of clusterings nearly the same amount, we opted to use RBR as the refinement scheme in *Nerstrand* due to its lower runtime.

D. Performance

Table III shows the mean and maximum modularity and the mean runtime for *s-Nerstrand* creating clusterings of the graphs from the 10th DIMACS Implementation Challenge. For comparison, the modularity and the runtime of the best clusterings (for which runtime was reported) from the challenge are shown on the right. While taking several orders of magnitude less time, *s-Nerstrand* finds clusterings with modularities equal to, or within a few percentage points of the best clusterings found in the challenge.

The high mean and maximum modularity of the clusterings generated by *s-Nerstrand* in 25 runs demonstrates the effectiveness of the multilevel paradigm for maximizing modularity.

TABLE III
COMPARISON TO DIMACS CHALLENGE RESULTS

	s-Nerstrand			DIMACS Best	
Graph	Mean	Max.	Time (s)	Max.	Time (s)
333SP	0.983	0.984	3.891	0.989	58614.5
G_n_pin_pout	0.480	0.485	1.105	0.500	3887.5
PGPgiantcompo	0.879	0.882	0.009	0.887	114.7
as-22july06	0.670	0.672	0.021	0.678	395.2
astro-ph	0.731	0.734	0.031	0.744	714.5
audikw1	0.913	0.914	3.484	0.917	75872.8
belgium.osm	0.993	0.993	0.771	0.995	6175.8
cage15	0.884	0.890	19.127	0.903	157390.2
caidaRouterLeve.	0.864	0.865	0.223	0.872	4859.2
celegans_metab.	0.442	0.446	0.001	0.452	4.4
citationCitesee.	0.817	0.819	0.467	0.824	4658.0
coAuthorsCitese.	0.895	0.896	0.327	0.905	5477.8
coPapersDBLP	0.855	0.857	2.024	0.867	36197.3
cond-mat-2005	0.729	0.733	0.066	0.746	2456.9
email	0.572	0.578	0.002	0.582	8.9
eu-2005	0.939	0.940	2.149	0.942	20488.6
in-2004	0.980	0.981	1.766	0.981	14639.0
ldoor	0.964	0.965	1.943	0.969	29138.2
luxembourg.osm	0.984	0.985	0.050	0.989	2453.6
memplus	0.689	0.694	0.020	0.700	193.6
polblogs	0.426	0.427	0.002	0.427	6.7
power	0.937	0.938	0.003	0.940	16.7
preferentialAtt.	0.276	0.278	1.241	0.302	81183.1
rgg_n_2_17_.	0.972	0.973	0.129	0.978	2251.7
smallworld	0.770	0.771	0.300	0.793	1007.8
uk-2002	0.990	0.990	45.028	0.990	478859.9

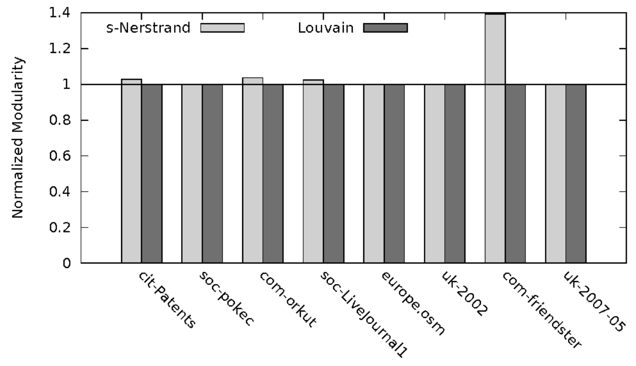

Fig. 8. The geometric mean modularity of clusterings generated by *s-Nerstrand* relative to *Louvain*.

By only aggregating vertices together for which the associated modularity gain is positive, we increase the lower bound on the quality of clusterings that can be generated at the coarsest levels. Because the coarse graph is of small size, we can afford to make many initial clusterings of it, and choose the best one, giving us a further guarantee on the quality of the clustering being generated. Finally, during during refinement we are able to continue to increase the modularity of the clustering at each level.

The quality of the clusterings generated by *s-Nerstrand* and *Louvain* are shown in Figure 8. In terms of clustering quality, *s-Nerstrand* generated clusterings with an average modularity equal to or slightly greater than *Louvain*. Across all eight graphs, *s-Nerstrand* produced clusterings that were on average 5.3% better. Although *s-Nerstrand* and *Louvain* produced clusterings of nearly identical (differing by less than 0.1%) modularity on *soc-pokec*, *europe.osm*, *uk-2002*, and *uk-2007-05*, *s-Nerstrand* produced clusterings with higher modularities for *cit-Patents*, *com-orkut*, *soc-LiveJournal1*, and *com-friendster*, at 2.7%, 3.7%, 2.4%, and 39.4% respectively. The high quality of clusterings being generated by *s-Nerstrand* despite its aggregation approach using only a single pass, is the result of the refinement performed on each of the coarse graphs. The significantly higher modularity of clusterings found by *s-Nerstrand* for com-friendster, is the result of *s-Nerstrand* being able to contract the graph down to ten vertices, whereas *Louvain* stopped at over 50 thousand and produced a much larger number of communities.

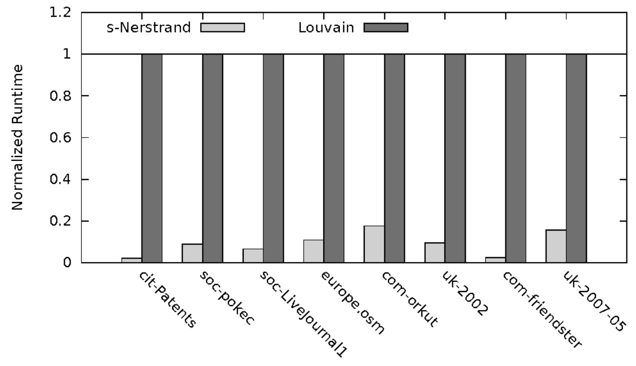

Fig. 9. The geometric mean runtime of *s-Nerstrand* relative to *Louvain*.

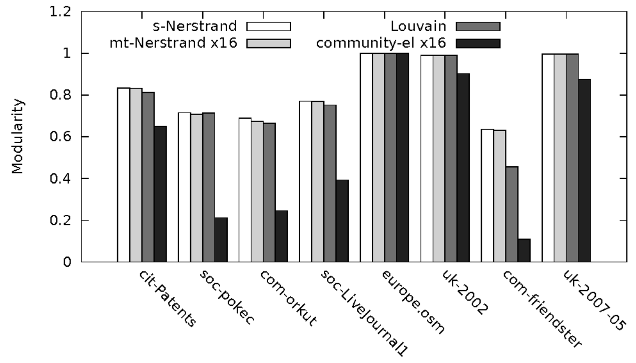

Fig. 10. The geometric mean modularity of generated clusterings.

The runtimes for generating clusterings for *s-Nerstrand* and *Louvain* are shown in Figure 9. *s-Nerstrand* outperformed *Louvain* for all graphs in this experiment in terms of computation time, and was on average 13.5 times faster. This difference in runtime can be attributed to the different ways in which aggregation is performed. In *s-Nerstrand*, each vertex is processed only once, whereas *Louvain* repeatedly processes its vertices until a local maxima in modularity is found.

IX. PARALLEL RESULTS

In this section we present the results of our experiments *mt-Nerstrand*. We show that not only does it achieve significant speedup over *s-Nerstrand* and outperform the competition, but does so without making sacrifices in terms of quality.

A. Quality

The effect on modularity of the parallelizing the serial algorithms in *s-Nerstrand* for *mt-Nerstrand* can be seen in Figure

10. We have included the results from *Louvain* and *community-el* for comparison. When run with 16 threads, *mt-Nerstrand* shows only minor degradation in cluster quality compared to its serial counter part, averaging 99.5% the modularity of *s-Nerstrand*. This is 4.8% higher modularity than clusterings produced by *Louvain*, and 89% higher than those produced by *community-el* using 16 threads. The reason *mt-Nerstrand* is able to produce clusterings with modularity similar to that of *s-Nerstrand* is that the quality of the coarsening and initial clustering phases is unaffected by the number of threads. It is not until the refinement step that we see a difference. This is the result of moves being made with stale cluster states. However, our results show that this has an extremely small effect on the quality.

B. Scaling

The runtimes of *mt-Nerstrand* and *community-el* using varying numbers of threads are shown in Table IV, along with those of the serial codes *s-Nerstrand* and *Louvain*. As can be seen, in both serial and parallel modes, *Nerstrand* produces clusterings extremely fast on all eight graphs, compared to *Louvain* and *community-el*.

The geometric mean of the speedups achieved by *mt-Nerstrand* with respect to *s-Nerstrand* for 16 threads was 6.2. The highest achieved speedup was 8.91 on the largest social network graph, *com-friendster*, and the lowest speedup of 5.15 was on the patent citation network, *cit-Patents*, which is also the smallest graph. We did not see as high of a speedup on this graph as a result of refinement performing extra work when done in parallel. For this graph, over twice as many refinement passes were made when using 16 threads as compared to when run serially.

The large amount of time taken by *community-el* on the uk-2002 and uk-2007-05 graphs can be explained by the presence of extremely high-degree vertices connected to mostly vertices of degree one. The aggregation in *community-el* is done via matching, and as a result takes several orders of magnitude more steps to contract these graphs than the other six.

The high performance of *mt-Nerstrand* comes from being based on the already fast algorithms of *s-Nerstrand*. During coarsening, one the most time intensive steps of the multilevel paradigm, *mt-Nerstrand* is able to use the same algorithm as *s-Nerstrand*, and scales well due to the unprotected grouping introduced in Section VI-B. The initial coarsening phase of *s-Nerstrand* is inherently parallel, and scales well when all of the threads can fit their data into the cache. Our parallel formulation of k-way boundary refinement with the order-independent updates described in Section VI-D, allows us to achieve high modularity in a scalably parallel fashion.

X. Conclusion

In this paper we presented several approaches to solving the issues associated with adapting the multilevel paradigm for maximizing modularity in serial and in parallel. We adapted the FirstChoice aggregation scheme from graph partitioning, such that is able to maximize modularity. We showed that this aggregation scheme works well for the modularity objective both in terms of quality and in terms of speed. We introduced a robust and fast method for generating clusterings of a contracted graph. We also introduced a modified version of the k-way boundary refinement for the modularity objective. We showed the combined computation complexity of these algorithms is $O(m+n)$. We then presented shared-memory parallel versions of these algorithms, which use sparse synchronization. This included a means of performing group-based aggregation effectively in parallel, and introducing an order independent method for updating cluster information during refinement without the use exclusive locks.

We presented these solutions in the form of the multi-threaded graph clustering tool *Nerstrand*, which is capable of producing high quality clusterings of large graphs extremely fast. We evaluated this tool on graph with millions vertices and billions of edges. Our tool finds clusterings of equal or better modularity than current speed oriented methods. *Nerstrand* is fast, finding these clusterings 2.7-44.9 times faster than competing methods.

Acknowledgment

This work was supported in part by NSF (IOS-0820730, IIS-0905220, OCI-1048018, CNS-1162405, and IIS-1247632) and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

[1] M. E. Newman, M. Girvan, Finding and evaluating community structure in networks, Physical review E 69 (2) (2004) 026113.

[2] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, D. Wagner, On modularity clustering, Knowledge and Data Engineering, IEEE Transactions on 20 (2) (2008) 172–188.

[3] M. E. Newman, Fast algorithm for detecting community structure in networks, Physical review E 69 (6) (2004) 066133.

[4] A. Clauset, M. E. Newman, C. Moore, Finding community structure in very large networks, Physical review E 70 (6) (2004) 066111.

[5] M. E. Newman, Modularity and community structure in networks, Proceedings of the National Academy of Sciences 103 (23) (2006) 8577–8582.

[6] D. A. Bader, K. Madduri, Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE, 2008, pp. 1–12.

[7] Y. Zhang, J. Wang, Y. Wang, L. Zhou, Parallel community detection on large networks with propinquity dynamics, in: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2009, pp. 997–1006.

[8] Ü. V. Catalyürek, K. Kaya, J. Langguth, B. Uçar, A divisive clustering technique for maximizing the modularity.

[9] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of Statistical Mechanics: Theory and Experiment 2008 (10) (2008) P10008.

[10] A. Noack, R. Rotta, Multi-level algorithms for modularity clustering, in: Experimental Algorithms, Springer, 2009, pp. 257–268.

[11] J. Riedy, D. A. Bader, H. Meyerhenke, Scalable multi-threaded community detection in social networks, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, IEEE, 2012, pp. 1619–1628.

[12] D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner (Eds.), Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings, Vol. 588 of Contemporary Mathematics, American Mathematical Society, 2013.

[13] S. Fortunato, Community detection in graphs, Physics Reports 486 (3) (2010) 75–174.

TABLE IV
THE GEOMETRIC MEAN OF RUNTIMES (IN SECONDS) OF THE SERIAL AND PARALLEL ALGORITHMS.

Threads	1	2	4	8	16
	s-Nerstrand	mt-Nerstrand			
cit-Paten.	13.9	10.2	5.6	3.6	2.7
soc-pokec	12.1	9.2	5.1	3.0	1.8
soc-LiveJ.	25.5	17.9	10.2	6.7	4.8
europe.osm	39.1	25.9	15.0	9.4	6.7
com-orkut	44.7	32.7	18.3	11.9	7.2
uk-2002	45.0	28.6	16.8	10.1	7.6
com-frien.	1864.0	1139.4	630.8	341.4	209.1
uk-2007-05	556.5	370.8	229.6	147.5	89.7
	community-el				
cit-Paten.	53.1	38.9	29.5	20.9	20.4
soc-pokec	33.8	26.5	14.8	8.7	5.9
soc-LiveJ.	115.1	87.5	51.3	31.7	25.9
europe.osm	46.3	43.6	30.8	24.3	20.0
com-orkut	181.3	135.4	72.5	41.6	26.8
uk-2002	6745.6	8519.0	6886.4	7242.3	7038.8
com-frien.	6959.7	4348.3	2256.9	1266.3	831.0
uk-2007-05	130675.0	150500.0	134750.0	104651.0	91193.4

[14] K. Wakita, T. Tsurumi, Finding community structure in a mega-scale social networking service, in: Proceedings of IADIS international conference on WWW/Internet 2007, 2007, pp. 153–162.

[15] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, P. Spyridonos, Community detection in social media, Data Mining and Knowledge Discovery 24 (3) (2012) 515–554.

[16] B. O. Fagginger Auer, R. H. Bisseling, Graph coarsening and clustering on the gpu, 10th DIMACS implementation challenge.

[17] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, Tech. Rep. SAND93-1301, Sandia National Labratories (1993).

[18] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, in: Proceedings of The International Conference on Parallel Processing, CRC PRESS, 1995, pp. III–113.

[19] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1996, Springer-Verlag, London, UK, UK, 1996, pp. 493–498.

[20] B. Monien, R. Diekmann, A local graph partitioning heuristic meeting bisection bounds, in: 8th SIAM Conf. on Parallel Processing for Scientific Computing, Vol. 525, Citeseer, 1997, p. 526.

[21] C. Walshaw, M. Cross, Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm, SIAM J. Sci. Comput. 22 (1) (2000) 63–80.

[22] P. Sanders, C. Schulz, Engineering multilevel graph partitioning algorithms, in: C. Demetrescu, M. Halldrsson (Eds.), Algorithms - ESA 2011, Vol. 6942 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 469–480.

[23] H. N. Djidjev, M. Onus, Scalable and accurate graph clustering and community structure detection, IEEE Transactions on Parallel and Distributed Systems.

[24] G. Karypis, V. Kumar, Multilevel k-way hypergraph partitioning, in: Proceedings of the 36th annual ACM/IEEE Design Automation Conference, ACM, 1999, pp. 343–348.

[25] A. Abou-Rjeili, G. Karypis, Multilevel algorithms for partitioning power-law graphs, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 2006, pp. 10–pp.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press, 2009, Ch. Priority queues.

[27] S. Fortunato, M. Barthelemy, Resolution limit in community detection, Proceedings of the National Academy of Sciences 104 (1) (2007) 36–41.

[28] M. Ovelgönne, A. Geyer-Schulz, An ensemble learning strategy for graph clustering, in: Graph Partitioning and Graph Clustering, American Mathematical Society, 2013, pp. 187–206.

[29] D. LaSalle, G. Karypis, Multi-threaded graph partitioning, in: Parallel & Distributed Processing Symposium (IPDPS), 2013 IEEE 27th International, IEEE, 2013.

[30] Ü. V. Catalyürek, M. Deveci, K. Kaya, B. Ucar, Multithreaded clustering for multi-level hypergraph partitioning, in: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, IEEE, 2012, pp. 848–859.

[31] J. Leskovec, J. Kleinberg, C. Faloutsos, Graphs over time: densification laws, shrinking diameters and possible explanations, in: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, ACM, 2005, pp. 177–187.

[32] L. Takac, M. Zabovsky, Data analysis in public social networks.

[33] L. Backstrom, D. Huttenlocher, J. Kleinberg, X. Lan, Group formation in large social networks: membership, growth, and evolution, in: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2006, pp. 44–54.

[34] J. Yang, J. Leskovec, Defining and evaluating network communities based on ground-truth, in: Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, ACM, 2012, p. 3.