# Multi-Threaded Modularity Based Graph Clustering using the Multilevel Paradigm

Dominique LaSalle and George Karypis

*Department of Computer Science & Engineering,*
*University of Minnesota,*
*Minneapolis, MN 55455, USA*

**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, which we show have a parallel runtime of $O(m/p+n/p+k)$, where $p$ is the number of threads and $k$ is the number of clusters. 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.

## 1. 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

*Corresponding author: Dominique LaSalle (lasalle@cs.umn.edu, 1-612-626-9873)
[1]The *Nerstrand* software is available at `http://cs.umn.edu/~lasalle/nerstrand`

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.

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 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 have $O(m+n)$ time and $O(m+n)$ space complexities. We show that the shared-memory parallel versions of these algorithms have a parallel time complexity of $O(m/p+n/p+k)$ where $p$ is the number of threads and $k$ is the number of clusters. To validate our contributions, we compare our implementation of these algorithms, *Nerstrand*, against the serial clustering tool *Louvain* [9] and the parallel clustering tools *community-el* [11] and *NetworKit* [12], and show that *Nerstrand* produces clusterings of equal or greater modularity and is 4.5–27.2 times faster than the methods that generate clusterings with competitive modularity. 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 2 we define the notation used throughout this paper. In Section 3 we give an overview of current graph clustering methods for maximizing modularity. In Section 4 we give an overview of the multilevel paradigm, its use in the graph partitioning problem, and more recently in the graph clustering problem. Descriptions of the serial algorithms we developed are presented in Section 5, and descriptions of their parallel counter parts are presented in Section 6. In Section 7 we describe our experimental setup. This is followed by the results of our experiments in Sections 8 and 9, in which we evaluate the quality and speed of the presented algorithms. Finally in Section 10, we review the findings of this paper.

## 2. 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 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

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

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

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

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.

## 3. Modularity Based Graph Clustering

A large number of approaches for maximizing modularity have been developed since it was first proposed [1] a decade ago. 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.

Although it does not maximize modularity explicitly, Label Propagation [15] is an iterative scheme that starts by assigning every vertex a unique label, and in every iteration a new label is assigned to each vertex based on the label of the majority of its neighbors. Although it does not maximize modularity as well many of the agglomerative schemes, its near linear running time still makes it an attractive option for maximizing modularity on large graphs.

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 recursively until a graph is generated in which no vertices change clusters. This is currently one of the fastest modularity based clustering methods available [16].

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 [17] 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.

Staudt and Meyerhenke [12] developed a parallel version of the Label Propagation algorithm. Their algorithm takes advantage of the independent nature of determining the label for each vertex, and as a result scales quite well. However, as is the case with the serial formulation of label propagation, it does not directly optimize modularity and can fail to produce clusterings with high modularity. Along with parallel label propagation, Staudt and Meyerhenke also proposed a parallel version of the Louvain method, which visits vertices in parallel and moves them between clusters using possibly stale cluster information. To further improve the quality of these clusterings, they also added a secondary move step (referred to as refinement) after the Louvain method has been recursively applied.

## 4. The Multilevel Paradigm

Multilevel methods for graph partitioning have been shown to be computationally efficient and lead to high-quality partitionings [18, 19, 20, 21, 22]. The multilevel paradigm is composed of three phases: *coarsening*, *initial clustering*, and *uncoarsening*.

A series of increasingly *coarser* (smaller) graphs, $G_1, \ldots, G_s$, is generated from the input graph $G_0$ in the coarsening phase. A solution $C_s$ of the smallest graph $G_s$ is generated in the initial clustering phase using a direct clustering algorithm (i.e., a non-multilevel clustering algorithm). Finally, in the uncoarsening phase, the initial solution is used to derive solutions for the successive *finer* (larger) graphs. This is done in two parts: first *projecting* the solution from $G_{i+1}$ to $G_i$, and then the solution is *refined*, making use of the increased degrees of freedom of the finer graph.

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 $\Delta Q$ 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.

## 5. 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:

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. The clustering $C$ is returned as *output*.

We introduce aggregation schemes to address the issue of coarsening graphs with power-law degree distributions in Section 5.1. We introduce a method for effectively generating initial clusterings of a coarsened graph in Section 5.2. We present a formulation of refinement for maximizing modularity in Section 5.3. We give a complexity analysis of these algorithms in Section 5.4, showing that they run in $O(m + n)$ time and $O(m + n)$ space.

### *5.1. Coarsening*

Coarsening is made up of two steps: *aggregation* and *contraction*. Aggregation is where matchings/groupings are assigned to each vertex in $G_i$, and in contraction, these matchings/groupings are used to generate the next coarser graph $G_{i+1}$. 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 all vertices, 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

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

In the special case where $v = u$, $Q_{merge}(v, u) = 0$ (i.e., there is no change in modularity if a vertex is merged with itself).

**Algorithm 1** Standard Matching (MAT)

```
1: function MATCH(G(V, E))
2:     Mark all v ∈ V as unmatched
3:     for all v ∈ V in random order do
4:         if v is unmatched then
5:             w ← v
6:             for all u ∈ Γ(v) do
7:                 if u is unmatched and Q_merge(v, u) > Q_merge(v, w) then
8:                     w ← u
9:                 end if
10:            end for
11:            Mark v and w as matched with each other
12:        end if
13:    end for
14: end function
```

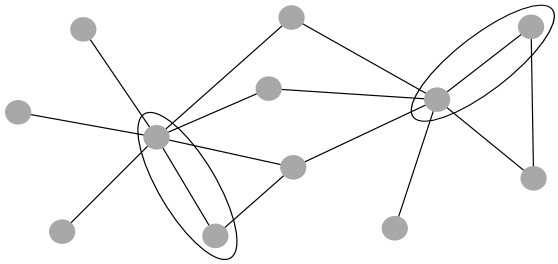


Figure 1: A small maximal matching.

*5.1.1. Matching*

Our standard matching algorithm (MAT) is outlined in Algorithm 1. It 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.

*5.1.2. 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). This matching scheme is outlined in Algorithm 2. As in MAT, all of $v$'s unmatched neighbors are searched to find the best vertex to match with (for brevity this is left out on line 3). If 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 $w \in \Gamma(u)$,

**Algorithm 2** Two-Hop Matching (M2M)

```
1: function TwoHopMatch(G(V, E))
2:     for all v ∈ V do
3:         ...
4:         if v is unmatched and d(v) ≤ max. two-hop degree then
5:             for all u ∈ Γ(v) in random order do
6:                 i ← 1
7:                 for all w ∈ Γ(u) do
8:                     if w is unmatched and d(w) ≤ max. two-hop degree then
9:                         Mark v and w as matched with each other
10:                        Break
11:                    end if
12:                    if i > max. neighbors to search then
13:                        Break
14:                    end if
15:                    i ← i + 1
16:                end for
17:            end for
18:        end if
19:    end for
20: end function
```

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. Additionally, we set a maximum number of neighbors that each vertex can search before it matches with itself. The neighbors of $u$ are searched in random order so as to not repeatedly search the parts of $u$ adjacency list. To ensure we do not decrease the quality of the matching, we limit two hop matching to only vertices with low degree.

*5.1.3. First Choice Grouping*

The third option we explored for aggregating power-law graphs was to allow more than two vertices to be aggregated together at a time. 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.

An outline of this scheme is given in Algorithm 3. 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

**Algorithm 3** First Choice Grouping (FCG)

```
1: function FirstChoiceGrouping(G(V, E))
2:     Mark all v ∈ V as unmatched
3:     for all v ∈ V do
4:         if v is unmatched then
5:             c ← an empty group
6:             for all u ∈ Γ(v) do
7:                 if u is unmatched then
8:                     if Q_merge(v, u) > Q_merge(v, c) then
9:                         c ← u
10:                    end if
11:                else
12:                    g ← group of u
13:                    if Q_merge(v, g) > Q_merge(v, c) then
14:                        c ← g
15:                    end if
16:                end if
17:            end for
18:            Add v to the group c
19:        end if
20:    end for
21: end function
```

**Algorithm 4** Initial Clustering

```
1: function InitialClustering(G(V, E))
2:     C(v) ← v for all v ∈ V
3:     Refine(C,G)
4:     return C
5: end function
```

is determined similarly, except the edges from $g$ to $v$ need to be summed, and $d(g)$ needs to be tracked.

### *5.2. Initial Clustering*

Once coarsening is finished, we are left with the coarsest graph $G_s$, and need to create the clustering $C = \{C_1, C_2, \ldots, C_k\}$. 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 setting each vertex to be a singleton cluster as in agglomerative clustering and then applying refinement as described in Section 5.3.2. This is similar to a single level of the Louvain method [9]. This is shown in Algorithm 4.

**Algorithm 5** Uncoarsening

```
1: function UNCOARSENING(G_i(V_i, E_i),C_{i+1})
2:     C_i ← projection of C_{i+1} onto G_i
3:     Refine(G_i(V_i, E_i),C_i)
4:     return C_i
5: end function
```

### *5.3. 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*.

#### *5.3.1. 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.

#### *5.3.2. 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 it 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

**Algorithm 6** Random Boundary Refinement

```
1: function RBR(G(V, E),C)
2:     repeat
3:         for all Boundary vertices v in random order do
4:             for all c ∈ clusters of Γ(v) do
5:                 if ΔQ(v, c) > ΔQ(v, C(v)) then
6:                     C(v) ← c
7:                 end if
8:             end for
9:         end for
10:    until No moves are made or max. # of iterations completed
11:    return C
12: end function
```

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 $C_i$ to cluster $C_j$ is given by the combined change in the cluster modularities:

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

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 (RBR) is described in Algorithm 6. It 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 (GBR) is described by Algorithm 7. It first inserts the boundary vertices into a priority queue. Each vertex is then extracted from this priority queue 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:

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

This however, is a computationally 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.

**Algorithm 7** Greedy Boundary Refinement

```
1: function GBR(G(V, E),C)
2:     repeat
3:         q ← PriorityQueue
4:         for all Boundary vertices v in random order do
5:             Insert v into q using equation (4)
6:         end for
7:         while q is not empty do
8:             for all c ∈ clusters of Γ(v) do
9:                 if ΔQ(v, c) > ΔQ(v, C(v)) then
10:                    C(v) ← c
11:                end if
12:            end for
13:            if v was moved then
14:                Update the priority of all u ∈ Γ(v) in q
15:            end if
16:        end while
17:    until No moves are made or max. # of iterations completed
18:    return C
19: end function
```

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.

When a vertex $v$ is moved from $C_i$ to $C_j$, we only update the priority of the vertices connected to it, even though all the priority of all vertices in $C_i$ and $C_j$ have changed. In our experiments we did not observe an increase in the modularity of clusterings if we kept the priority of all vertices up to date.

### *5.4. Complexity Analysis*

The overall complexity for the serial algorithms in *Nerstrand* is the sum of its three phases:

1. Coarsening, $O(m + n)$, in Section 5.4.2.
2. Initial Clustering, $O(m + n)$, in Section 5.4.3.
3. Uncoarsening, $O(m + n)$ for RBR and $O(m \log n)$ for GBR, in Section 5.4.4.

Adding these we get an overall computational complexity of $O(m + n)$ (and $O(m \log n)$ if GBR is used), where $m$ is the number of edges and $n$ is the number of vertices. The space complexity is determined by the combined size of the generated graphs, which we show to be $O(m + n)$ in Section 5.4.1.

*5.4.1. 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 $n^2/2$ and $m^2/2$ respectively giving a computational and space 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:

$$\sum_{i=0}^{s} |G_i| = \sum_{i=0}^{s} |G_0|\alpha^i = |G_0|\frac{1-\alpha^{s+1}}{1-\alpha}. \tag{5}$$

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 $\alpha$ is a constant, we can see then that the total number of vertices is $O(n)$ and the total number of edges is $O(m)$. That is, $\sum_{i=0}^{s} n_i = O(n)$ and $\sum_{i=0}^{s} m_i = O(m)$. Our choice of $\alpha$ 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.

*5.4.2. 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 (Algorithm 2, line 12).

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. Since we established that $\sum_{i=0}^{s} n_i = O(n)$ and $\sum_{i=0}^{s} m_i = O(m)$ in Section 5.4.1, we can then say that the coarsening phase takes $O(m + n)$ time.

*5.4.3. 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. Setting each vertex to be a singleton cluster takes $O(1)$ time per vertex, and thus $O(n_s)$ time total, and $O(n_s)$ space for the cluster labels. Then, performing a pass of Random Boundary Refinement on the $n_s$ clusters takes $O(n_s + m_s)$ time as described in Section 5.4.4. A constant number of clusterings are created, so in total the complexity of initial clustering is $O(n_s + m_s)$. The only bounds on the size of the input graph for initial clustering is $n_s \leq n$ and $m_s \leq m$, thus the complexity of initial clustering is bounded by $O(n + m)$.

*5.4.4. Uncoarsening Complexity*

Projection is a simple lookup in two arrays for each vertex in the fine graph $G_i$, thus 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.

GBR performs the same operations as RBR 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 graphs in the

multilevel hierarchy 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 using replacement we can see that the total complexity of GBR is $O(m \log n)$.

## 6. 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 methods for assigning vertices to threads in a manner that balances the number of edges for which a thread is responsible. 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 boundary refinement in parallel for the modularity objective.

Our general approach to parallelization follows a coarse-grained model, where threads allocate their own memory and synchronization points are minimized [27]. 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.

### *6.1. Graph Distribution*

For distributing vertices and their associated edges among threads, we experimented with three strategies. All three strategies balance the number of edges assigned to each thread, as this is the dominating factor in the runtime.

The first strategy, which we will refer to as a *block* distribution, preserves the original ordering of the vertices of the graph, and assigns a continuous chunk of vertices to each thread such that the sum of the degrees is roughly $2m/p$, where $p$ is the number of threads. This strategy has the benefit of preserving memory friendly orderings if they exist. However, it can lead to significantly different numbers of vertices being assigned to threads if the vertex degrees are not evenly distributed in the original ordering.

The second strategy, which we will refer to as a *cyclic* distribution, permutes the vertex order in a cyclic fashion using cycles of size $p$. That is, the array's indices will be reordered to $\{1, p+1, 2p+1, \ldots, 2, p+2, 2p+2, \ldots\}$. Then, each thread is assigned a continuous chunk of permuted vertices such that the sum of the degrees is roughly $2m/p$. Note that this is different from a traditional cyclic distribution in that a thread may be assigned vertices from multiple cycles. This strategy has the benefit of leading to a more even vertex distribution when the vertex degrees of the original ordering are not evenly distributed. However, it sacrifices the benefits of orderings that are memory friendly.

**Algorithm 8** Parallel Matching Cleanup

```
1: function CLEANUP(G(V, E),M)
2:     Synchronize threads
3:     T ← all vertices owned by this thread
4:     for all v ∈ T do
5:         if M(M(v)) ≠ v then
6:             M(v) ← v
7:         end if
8:     end for
9:     return M
10: end function
```

The third strategy, which we will refer to as a *block-cyclic* distribution, attempts to combine the best of both of these strategies. It permutes the vertex order in a block-cyclic fashion. That is, the array's indices will be reordered to $\{B_1, B_{p+1}, B_{2p+1}, \ldots, B_2, B_{p+2}, B_{2p+2}, \ldots\}$, where $B_i$ is the $i$th block of vertices. Using larger blocks will mean more of the original ordering will be preserved and possibly any memory friendly properties, but will increase the likely hood that the vertices will not be balanced among threads. Using smaller blocks will have the opposite effect.

### *6.2. 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 vertex that has been matched with $v$. For example if we have a graph with five vertices, and vertices 1 and 5 are matched together, vertices 3 and 4 are matched together, and vertex 2 is matched with itself, then the matching vector $M$ would be $\{5, 2, 4, 3, 1\}$.

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 $w$. The cleanup process for fixing these broken matchings is described in Algorithm 8. 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, $M(v) = v$ (line 6). This technique for performing parallel matching was first proposed by Catalyürek et al. [28], 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 parallel method for grouping vertices in an unprotected fashion. 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 $M(w) = v$.

To accomplish this during aggregation, all vertices are initially grouped 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 an invalid vertex number being written), a simple validity check can be added. Vertices that are part of a tail are not part of a valid grouping, and must be cleaned up.

In order to cleanup these tails, the following method described in Algorithm 9 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 using a hash table (line 10). 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).

Contraction is an inherently parallel process, as for any matching or group of vertices being collapsed, the creation of the resulting coarse vertex and coarse edges depends only upon the finer graph $G_i$ and matching/grouping vector $M$. Threads are responsible for contracting the matchings/groups of fine vertices which form the coarse vertices they will own.

### *6.3. 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 as a singleton 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, each thread initialize the vertices it owns to singleton clusters, and then the threads work cooperatively to create each initial clustering using the parallel formulation of refinement described below in Section 6.4.

**Algorithm 9** Parallel Group Cleanup

```
1: function GroupCleanup(G(V, E),M)
2:     T ← all vertices owned by this thread
3:     Mark all v ∈ T as not finalized
4:     for all v ∈ T do
5:         if v is not finalized then
6:             H ← a HashTable
7:             u ← v
8:             for i ← 1 to max group size do
9:                 u ← M(v)
10:                if u ∈ H then
11:                    if u = v then
12:                        if Minimum w ∈ H is owned by this thread then
13:                            Mark all w ∈ H as finalized
14:                        end if
15:                    else
16:                        M(v) ← v and mark v as finalized
17:                    end if
18:                    Break
19:                else
20:                    Insert u into H
21:                end if
22:            end for
23:            if v has not been finalized then
24:                M(v) ← v and mark v as finalized
25:            end if
26:        end if
27:    end for
28: end function
```

### *6.4. 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.

Our parallel refinement algorithm is described in Algorithm 10. Instead of using locking clusters, each thread 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 (line 12). Because each thread is unaware of the moves being made by

**Algorithm 10** Parallel Random Boundary Refinement

```
1: function ParRBR(G(V,E),C)
2:     repeat
3:         C' ← C.
4:         D ← empty List
5:         for all Boundary vertices v this thread owns in random order do
6:             for all c ∈ clusters of Γ(v) do
7:                 if ΔQ(v,c) > ΔQ(v,C'(v)) then
8:                     C'(v) ← c
9:                 end if
10:            end for
11:            if v was moved then
12:                Add move to D and apply local updates to C'
13:            end if
14:        end for
15:        Synchronize threads
16:        C' ← prospective changes from all threads
17:        for all m ∈ D in reverse order do
18:            if ΔQ(v,c) ≤ 0 then
19:                Remove m from D and rollback local updates to C'
20:            end if
21:        end for
22:        Synchronize threads
23:        Apply remote updates to C' and reduce to C
24:        Synchronize threads
25:    until No moves are made or max. # of iterations completed
26:    return C
27: end function
```

other threads, a move that it sees as 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. This can be see on line 17. 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 $\{v,u\}$.

For Random Boundary Refinement, each thread visits the boundary vertices it owns in random order. 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.

### *6.5. Parallel Complexity*

The overall complexity for the parallel algorithms in *Nerstrand* is the sum of its three phases:

1. Coarsening, $O(m/p + n/p)$, in Section 6.5.1.
2. Initial clustering, $O(m/p + n/p + k)$, in Section 6.5.2.
3. Uncoarsening, $O(m/p + n/p + k)$ for RBR and $O((m/p)\log(n/p) + k)$ for GBR, in Section 6.5.3.

Adding these we get an overall parallel complexity of $O(m/p + n/p + k)$ (and $O((m/p)\log(n/p)+k)$ if GBR is used), where $m$ is the number of edges, $n$ is the number of vertices, $p$ is the number of threads, and $k$ is the number of clusters.

#### *6.5.1. Coarsening Complexity*

When using MAT or M2M to coarsen a graph in parallel, each thread is responsible for finding matches for its set of vertices, which entails scanning all incident edges, which makes finding matches for all vertices an $O(m/p + n/p)$ operation. Then, to fix the broken matchings, each thread makes a second cleanup pass over its vertices, which is an $O(n/p)$ operation.

For FCG, it takes $O(m/p + n/p)$ time for threads to group their sets of vertices. Performing the group cleanup requires each thread to iterate over each of its vertices. Then, for each vertex, the inner loop on line 8 of Algorithm 9 can search up to the maximum group size number of vertices in the worst case.

This gives group cleanup an upper bound on runtime of $O(n/p)$, albeit with a very large constant in front (the maximum group size). Even when many of the groupings reach maximum size, this rarely plays a significant factor in the total runtime as the edges in a graph usually greatly outnumber the vertices, and once the owning thread makes a pass over the group, the vertices will get marked as finalized and will not be traversed again.

Contraction requires each thread to iterate over the fine vertices and edges that form the coarse vertices and edges it will own in the next graph. This gives contraction a parallel time complexity of $O(m/p + n/p)$.

Putting all of the parallel complexities from coarsening together, we that parallel complexity of coarsening is $O(m/p + n/p) + O(n/p) = O(m/p + n/p)$.

*6.5.2. Initial Clustering Complexity*

For small numbers of threads and small coarsest graphs, each thread independently generates and initial clustering in $O(m + n)$ time. However, for large numbers of threads or large coarsest graphs, the initial clusterings are generated cooperatively. Which means that each thread initialize its own $n/p$ vertices to singleton clusters. Then parallel RBR is applied to the singleton clusters, which is shown in the next section to have a parallel complexity of $O(m/p + n/p + k)$ for $m$ edges and $n$ vertices.

*6.5.3. Uncoarsening Complexity*

Projection is an inherently parallel process where each thread projects the clusters from the vertices in the coarse graph to the $n/p$ fine vertices it owns. This results in $O(n/p)$ time.

For parallel RBR, each thread iterates over its own at most $n/p$ boundary vertices and considers them for moving as in serial RBR. Using the $O(m + n)$ result for serial RBR in Section 5.4.4 for these $n/p$ vertices with $m/p$ incident edges, we then know that the movement pass of parallel RBR takes $O(m/p+n/p)$ time. The additional roll-back pass performs at most the same operations, and thus is also bounded by $O(m/p + n/p)$ time.

Then in the two steps where the state of the clusters is updated, at lines 16 and 23 in Algorithm 10, all of the $p$ threads states for each of the $k$ clusters must be combined. This can be done by assigning $k/p$ clusters to each thread and then having the assigned thread combine the $p$ values for each of the clusters it is assigned. This results in a time complexity of $O(pk/p) = O(k)$. Thus parallel RBR takes $O(m/p + n/p + k)$ time. At the coarser levels where $k$ is close to $n$, we can see that it dominates the runtime. Then, at the finer levels where $k$ is much smaller than $n$, the $O(m/p + n/p)$ term dominates the runtime.

For parallel GBR, each thread has the extra work of maintaining its priority queue which could contain up to $n/p$ vertices, and make up to $m/p$ updates per iteration. This plus the $O(m/p + n/p)$ for the rollback pass, and $O(k)$ for the cluster state updates makes the parallel complexity for GBR $O((m/p) \log(n/p) + k)$.

Table 1: Graphs Used in Experiments

| Graph | # Vertices | # Edges |
|---|---|---|
| cit-Patents[29] | 3,774,768 | 16,518,947 |
| soc-pokec[30] | 1,632,803 | 22,301,964 |
| soc-LiveJournal1[31] | 4,846,609 | 42,851,237 |
| europe.osm[32] | 50,912,018 | 54,054,660 |
| com-orkut[33] | 3,072,441 | 117,185,083 |
| uk-2002[32] | 18,520,486 | 261,787,258 |
| com-friendster[33] | 65,608,366 | 1,806,067,135 |
| uk-2007-05[32] | 105,896,555 | 3,301,876,564 |

Putting these results together with the $O(n/p)$ time for projection, we can see that parallel uncoarsening takes $O(m/p + n/p + k)$ time for RBR, and $O((m/p)\log(n/p))$ time for GBR.

## 7. Experimental Methodology

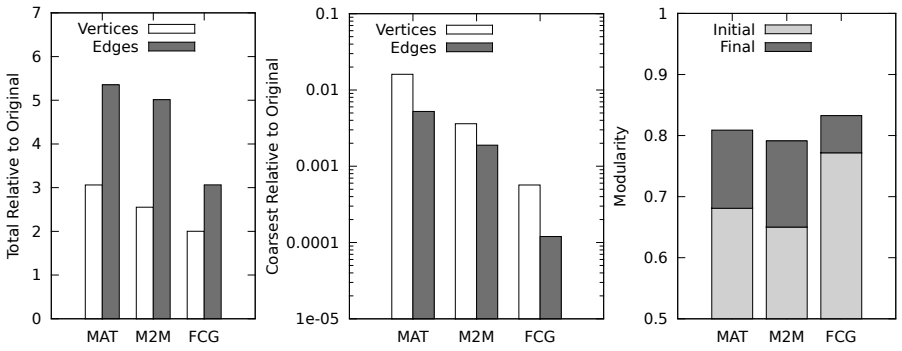
The experiments that follow were run on an HP ProLiant BL280c G6 with 2x 8-core Xeon E5-2670 @ 2.6 GHz system with 256GB of memory. We used GCC 4.7 and the accompanying libgomp that conforms to the OpenMP 3.1 specification. Unless otherwise noted, all runs were repeated 25 times with different random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

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 [16] available serial method for modularity maximization on large graphs, *Louvain* [9]. We used version 0.2 which is available from `https://sites.google.com/site/findcommunities/`. Because of *Nerstrand*'s similarity with multilevel graph partitioners, we compare against *Metis* [18] using version 5.1.0, available from `http://cs.umn.edu/~metis`. To facilitate finding clusters, we allowed for up to a 50000% imbalance and for the number of partitions we used powers of two from eight to $16,384$, and selected the clustering/partitioning that resulted in the highest modularity.

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/`, and the implementations of parallel label propagation (*PLP*) and the parallel louvain method with refinement (*PLMR*) provided by NetworKit [12] using version 3.1 available at `https://networkit.iti.kit.edu/`.

Table 1 shows the graphs primarily used for evaluation in Sections 8 and 9. Some of these are directed graphs, but for these experiments we created undi-




(a) (b) (c)

Figure 2: The total number of edges and vertices generated during the multilevel process relative to that of the input graph (a), the size of the coarsest graph relative to the input graph (b), and the mean modularity (c), for each coarsening scheme.

rected versions to be compatible with the modularity objective. We also used graphs from the 10th DIMACS Implementation Challenge [32]. This challenge consisted of two parts, graph partitioning and graph clustering. Participants submitted algorithms and subsequent implementations to compete in several categories, one of which was modularity maximization.

## 8. Serial Results

Sections 8.1 through 8.3 present the results of our experiments designed to evaluate the different schemes for coarsening, initial clustering, and uncoarsening. For these evaluations we used the graphs described by Table 1, with the exception of com-friendster and uk-2007-05, which were excluded due to their size.

In Section 8.4 we present 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 1 were used for the comparison.

### *8.1. Aggregation Schemes*

We evaluated the three aggregation schemes (MAT, M2M, and FCG) using three criteria: rate of contraction, size of coarsest graph, and effect on modularity.

#### *8.1.1. Rate of Contraction*

We measured the rate of contraction by using the total number of vertices and edges found in $G_0$ through $G_s$, as this directly correlates to the amount

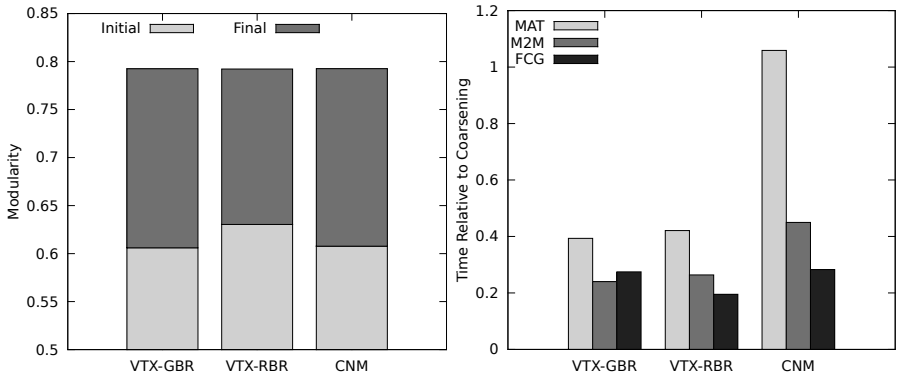
of work done in the coarsening and uncoarsening phases, and the total amount of space used. This is shown in Figure 2a. The plain vertex matching scheme, MAT, did the worst, on average generating a total of 3.1 times as many vertices and 5.4 times as many edges as in the original graph. M2M did better, 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. This shows that while using the 0.95 minimum coarsening rate allows for up to 20 times as much space required as the size of the input graph as shown in Section 5.4, in practice for FCG it is closer to only 3 times as much space as required by the original graph.

*8.1.2. Size of Coarsest Graph*

Figure 2b shows the number of vertices/edges in the coarsest graph divided by the number of vertices/edges in the original graph ($y$-axis is in log-scale). As expected, M2M outperformed MAT generating a coarsest graph of roughly half the size on average, a result of a additional vertices being matched with two-hop neighbors. FCG greatly outperformed the other methods averaging a coarsest graph with an order of magnitude less vertices and two orders of magnitudes less 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.

*8.1.3. Effect on Modularity*

Figure 2c shows the modularity after the initial clustering of the coarsest graph, as well as the final modularity of the 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. 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. Just as FCG did the best at creating




(a) (b)

Figure 3: The mean modularity (a) and the mean runtime relative to coarsening (b) of the initial clustering schemes.

a sparse coarse graph, it also resulted in the highest average modularity, of 0.832. This is again due to FCG always choosing the highest gain merges.

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*.

### *8.2. 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 (VTX-RBR). We ran VTX-RBR with 16 different random seeds to generate different initial clusterings and selected the best one. 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 3a 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 these clusterings were projected and refined to the original graphs, all three schemes had a final modularity of 0.792. This show the power of the multilevel paradigm, where a large amount of the solution quality is a result of coarsening, which was performed identically for all three schemes, and in refinement where rough solutions can often be significantly improved.

Figure 3b 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. While on finer graphs GBR exhibits near linear runtimes as only a fraction of the vertices are on the boundary, VTX-GBR starts with all vertices on the boundary, thus VTX-GBR performs very close to its worse case runtime of $O(m \log n)$. 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*.

### *8.3. Refinement Schemes*

Table 2: 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 2. 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.

### *8.4. Performance*

The quality of the clusterings generated by *s-Nerstrand* and *Metis* relative to *Louvain* are shown in Figure 4. 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

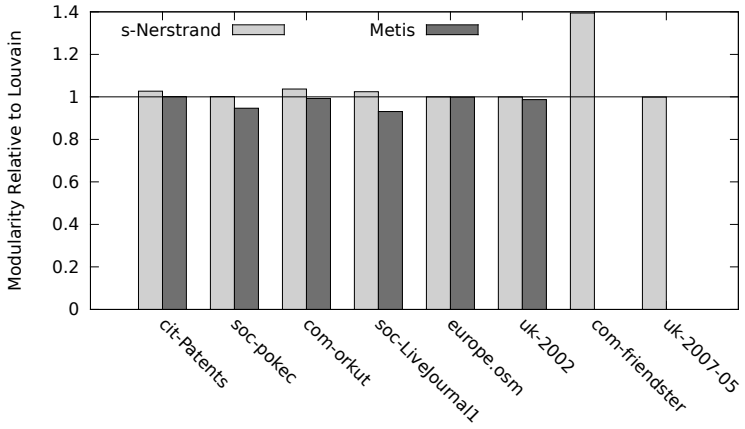



Figure 4: The modularity of clusterings generated by *s-Nerstrand* relative to *Louvain*.

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.

Due to its slower rate of contraction, *Metis* was unable to cluster the two largest graphs in the 256GB of memory in our test machine. *Metis* is able to produce clusterings with modularities that are within 1–2% of *s-Nerstrand* for graphs with strong community structure (europe.osm and uk-2002), the edgecut and modularity objectives both find areas of extremely low connectivity to place cluster boundaries. However, for graphs with less strong community structures (cit-Patents, soc-pokec, com-orkut, and soc-LiveJournal1), *Metis* produces clusterings that are 3–10% lower than *s-Nerstrand* as the two objectives diverge. In addition to this, the number of clusters must be known a priori for the algorithms in *Metis*.

The runtimes for generating clusterings for *s-Nerstrand* and *Metis* relative to *Louvain* are shown in Figure 5. *s-Nerstrand* outperformed *Metis* and *Louvain* for all graphs in this experiment in terms of computation time, and was 1.04–4.25 times faster than *Metis* and 5.66-44.9 times faster than *Louvain*. The lower runtime of *s-Nerstrand* than *Metis* is the result of its superior contraction rate made possible by FCG aggregation that groups many vertices together at a time while decreasing the edge density in resulting graphs. This difference in runtime between *s-Nerstrand* and *Louvain* 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

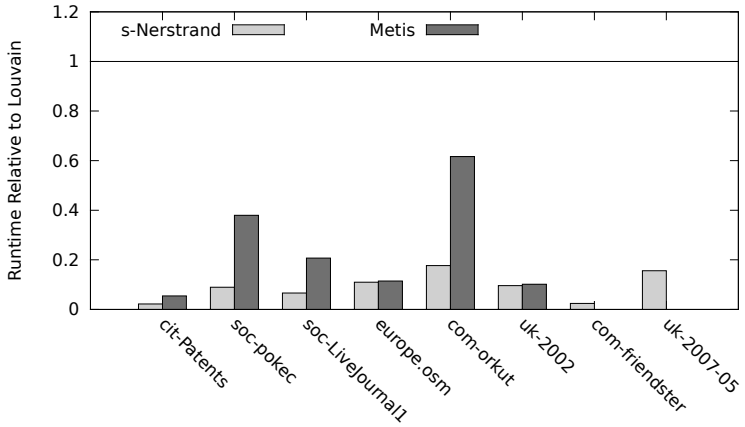



Figure 5: The runtime of *s-Nerstrand* relative to *Louvain*.

modularity is found.

The scaling of *s-Nerstrand* with respect to the number of edges in the input graph is shown in Figure 6, with 25 graphs from the DIMACS Challenge [32] in addition to the eight listed Table 1. A line has been fitted to these point to show their trend, with a slope of 183 nanoseconds per edge (or 55 million edges per second). This shows for real world datasets *s-Nerstrand* demonstrates linear scalability with a very small constant factor.

## 9. Parallel Results

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

### *9.1. Graph Distribution*

The effects on runtime of the different graph distribution strategies is shown in Figure 7. The block-cyclic distribution was run with a block size of $4,096$. Concerning the performance difference between a block distribution and a cyclic distribution, we see an even split where the block distribution performs better for half of the graphs and the cyclic distribution performs better for the other half.

Overall, the block-cyclic distribution performed the best, being the fastest distribution on five of the eight graphs, and on the three graphs where it was not the fastest, it was second, showing its robustness as a distribution strategy. This is because it combines the memory friendly ordering properties of the block distribution and the load balancing properties of the cyclic distribution. For the case where the block-cyclic distribution performed worse than the block distribution, uk-2007-05, block-cyclic was as fast or faster in all steps except the

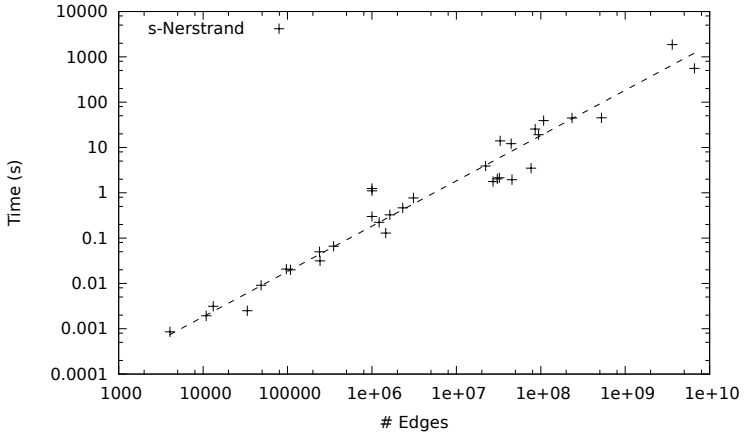



Figure 6: The scaling of *s-Nerstrand* with respect to the number of edges in a graph.

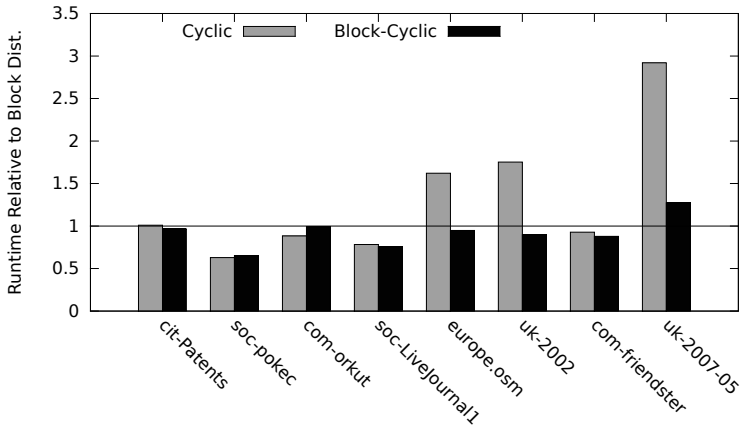



Figure 7: The runtimes of each distribution using 16 threads relative to a block distribution.

most memory intensive step, contraction, where it was just over twice as slow. However, if we increase the block size from $4,096$ to $16,384$, the block-cyclic distribution becomes faster than the block distribution on this graph.

Where all three distribution schemes balance the number of edges across the threads, the ratio of the maximum number of vertices to the average number assigned to a thread, the vertex imbalance, was highest for the block distribution. The block distribution averaged a vertex imbalance of 4.60 for the eight graphs, and was highest on uk-2007-05 at 9.12. The cyclic and block-cyclic distributions both averaged vertex imbalances of 1.36, and also had their highest vertex imbalances on uk-2007-05 at 1.61 and 1.63 respectively.

Due to the block-cyclic distribution's superior performance overall, it is the

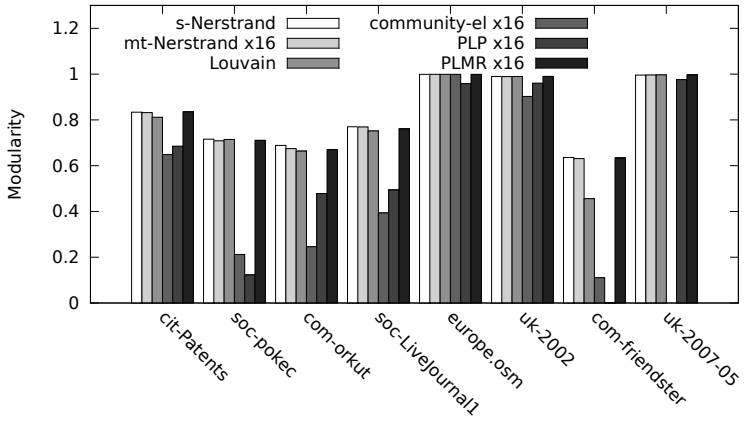



Figure 8: The modularity of generated clusterings of the different algorithms.

distribution used by *Nerstrand* in the experiments that follow (continuing to use a block-size of $4,096$).

### *9.2. Quality*

The effect on modularity of the parallelizing the serial algorithms in *s-Nerstrand* for *mt-Nerstrand* can be seen in Figure 8. We have included the results from *Louvain*, *community-el*, *PLP*, and *PLMR* 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*. Compared to other parallel methods using 16 threads, *mt-Nerstrand* produced clusterings with 89% higher modularity than *community-el*, and 215% higher than those produced by *PLP*. The clusterings produced by *PLMR* were of near identical modularity to *mt-Nerstrand*, with *mt-Nerstrand* producing clusterings of only 0.07% higher modularity.

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 partially stale cluster states. However, our results show that this has an extremely small effect on the quality.

The low quality of the clusterings produced by *PLP*, particularly on the social network graphs which tend to have higher inter-cluster connectivity, is due to it not directly optimizing modularity. However, on the web graphs and the citation network where clusters have low inter-cluster connectivity, it was able to find clusterings of modularity within a few percentage points of those found by *mt-Nerstrand*.

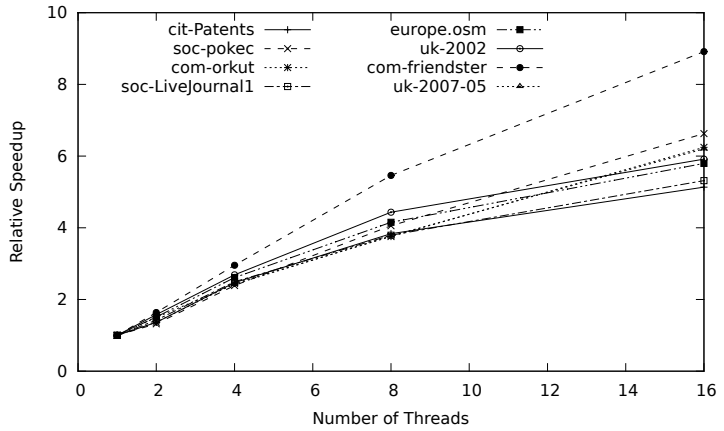



Figure 9: The parallel scaling of *mt-Nerstrand* for the eight test graphs.

*9.3. Scaling*

The speedups achieved by *mt-Nerstrand* with respect to *s-Nerstrand* are shown in Figure 9. The mean speedup for all eight graphs using 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 $k$ component of the $O(m/p + n/p + k)$ parallel complexity of *mt-Nerstrand* played relatively little role in the scaling, as its largest value was for the uk-2007-05, at 760 thousand, far below the 3.3 billion edges and 105 million vertices. The smallest value for $k$ was on com-friendster, at nine.

Figure 10 shows the runtimes of *mt-Nerstrand*, *community-el*[2], *PLP*, and *PLMR*, using 16 threads. The runtime relative to *mt-Nerstrand* is represented by the height of each bar, while the absolute runtime in seconds is displayed at the top of each bar.

The only method faster than *mt-Nerstrand* was *PLP*. Despite the added overheads of using the multilevel paradigm which allows it to find clusterings of significantly higher modularity, *mt-Nerstrand* is only on average 42% slower than *PLP* using 16 threads, up to 204% slower for com-friendster, and for europe.osm *mt-Nerstrand* was 142% faster. This high variability in runtime relative to the size of the graph for *PLP* is due the number of iterations it takes for the labels to fully propagate. For com-friendster it took seven iterations on average to find a clustering solution, whereas for europe.osm it took 546 iterations on average to

[2]Timings for *community-el* on the uk-2002 and uk-2007-05 graphs were performed with its *coverage* option to terminate the runs early (set to 75% and 50% respectively).

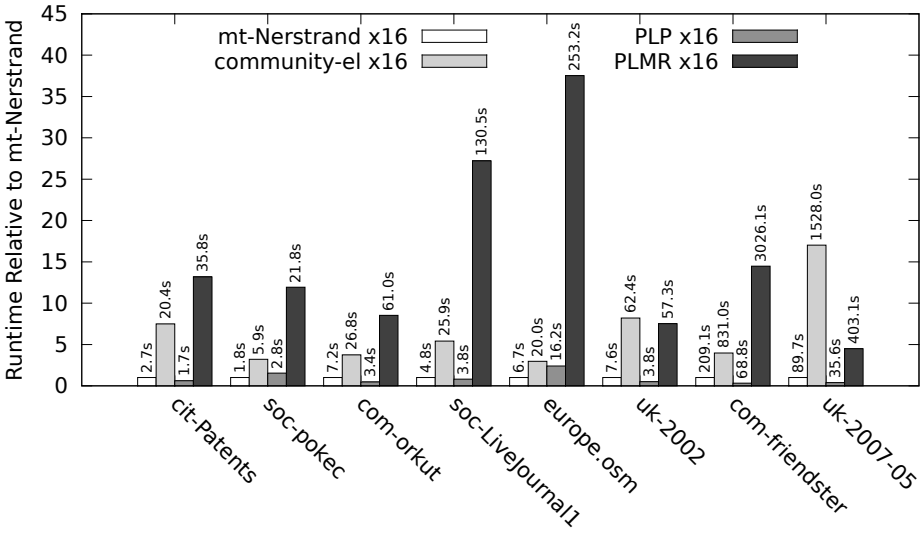



Figure 10: The runtimes of the parallel clustering methods relative to *mt-Nerstrand*, with their absolute runtimes listed above.

find a clustering solution. This expounds one of the strengths of the multilevel paradigm. Where label propagation takes $O(\delta)$ iterations to propagate through a cluster with a diameter of $\delta$, the vertex contraction of *mt-Nerstrand* takes only roughly $O(\log(\delta))$ levels to fully contract the cluster as groups of vertices are recursively merged together.

The high parallel 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 6.2. 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 boundary refinement with the order-independent updates described in Section 6.4, allows us to achieve high modularity in a scalably parallel fashion.

## 10. 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 followed these with a modified version

of boundary refinement for the modularity objective. We showed the combined computational complexity of these algorithms is $O(m+n)$. We then presented shared-memory parallel versions of these algorithms. 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 showed that these shared-memory parallel algorithms have parallel complexity of $O(m/p+n/p+k)$ where $p$ is the number of threads and $k$ is the number of clusters, and achieve speedups of 5.1–8.9 over their serial counterparts.

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 methods. *Nerstrand* is fast, finding these clusterings 4.5–27.2 times faster than competing methods that produce results of similar quality.

**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.

[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] C. L. Staudt, H. Meyerhenke, Engineering high-performance community detection heuristics for massive graphs, in: proceedings of the 2013 International Conference on Parallel Processing, Conference Publishing Services (CPS), 2013.

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

[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] U. N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Physical Review E 76 (3) (2007) 036106.

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

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

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

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.