Graph Coarsening and Clustering on the GPU

B. O. Fagginger Auer and R. H. Bisseling

Mathematics Institute, Utrecht University,
Budapestlaan 6, 3584 CD, Utrecht, the Netherlands
B.O.FaggingerAuer@uu.nl
R.H.Bisseling@uu.nl

Abstract. Agglomerative clustering is an effective greedy way to quickly generate graph clusterings of high modularity in a small amount of time. In an effort to use the power offered by multi-core CPU and GPU hardware to solve the clustering problem, we introduce a fine-grained shared-memory parallel graph coarsening algorithm and use this to implement a parallel agglomerative clustering heuristic on both the CPU and the GPU. This heuristic is able to generate clusterings in very little time: a modularity 0.996 clustering is obtained from a street network graph with 14 million vertices and 17 million edges in 4.6 seconds on the GPU.

1 Introduction

We present a fine-grained shared-memory parallel algorithm for graph coarsening and apply this algorithm in the context of graph clustering to obtain a fast greedy heuristic for maximising modularity in weighted undirected graphs. This is a follow-up to [7], which was concerned with generating weighted graph matchings on the GPU, in an effort to use the parallel processing power offered by multi-core CPUs and GPUs for discrete computing tasks, such as partitioning and clustering of graphs and hypergraphs. Just as generating graph matchings, graph coarsening is an essential aspect of both graph partitioning [4,8,11] and multi-level clustering [21] and therefore forms a logical continuation of the research done in [7].

Our contribution is a parallel greedy clustering algorithm, that scales well with the number of available processor cores, and generates clusterings of reasonable quality in very little time. We have tested this algorithm, see Sec. 5, against a large set of clustering problems, all part of the 10th DIMACS challenge on graph partitioning and clustering [1], such that the performance of our algorithm can directly be compared with the state-of-the-art clustering algorithms participating in this challenge.

An *undirected graph* G is a pair (V, E), with vertices V, and edges E that are of the form $\{u, v\}$ for $u, v \in V$ with possibly $u = v$. Edges can be provided with weights $\omega : E \to \mathbf{R}_{>0}$, in which case we call G a *weighted undirected graph*. For vertices $v \in V$, we denote the set of all of v's *neighbours* and v's *degree* by

$$V_v := \{u \in V \mid \{u, v\} \in E\} \setminus \{v\} \qquad \text{and} \qquad \deg(v) := |V_v|.$$

A *matching* of $G = (V, E)$ is a subset $M \subseteq E$ of the edges of G, satisfying that any two edges in the matching are disjoint. We call a matching M *maximal* if there does not exist a matching M' of G with $M \subsetneq M'$ and we call it *perfect* if $2\,|M| = |V|$. If $G = (V, E, \omega)$ is weighted, then the *weight* of a matching M of G is defined as the sum of the weights of all edges in the matching: $\omega(M) := \sum_{e \in M} \omega(e)$. A matching M of G which satisfies $\omega(M) \geq \omega(M')$ for every matching M' of G is called a *maximum-weight matching*.

Clustering is concerned with partitioning the vertices of a given graph into sets consisting of vertices related to each other, e.g. to isolate communities in graphs representing large social networks [2,13]. Formally, a *clustering* of an undirected graph G is a collection $\mathcal{C}$ of subsets of V, where elements $C \in \mathcal{C}$ are called *clusters*, that forms a partition of G's vertices, i.e.

$$V = \bigcup_{C \in \mathcal{C}} C, \qquad \text{as a disjoint union.}$$

Note that the number of clusters is not fixed beforehand, and that there can be a single large cluster, or as many clusters as there are vertices, or any number of clusters in between. A quality measure for clusterings, *modularity*, was introduced in [15], which we will use to judge the quality of the generated clusterings.

Let $G = (V, E, \omega)$ be a weighted undirected graph. We define the weight $\zeta(v)$ of a vertex $v \in V$ in terms of the weights of the edges incident to this vertex as

$$\zeta(v) := \begin{cases} \sum\limits_{\{u,v\} \in E} \omega(\{u, v\}) & \{v, v\} \notin E, \\ \sum\limits_{\substack{\{u,v\} \in E \\ u \neq v}} \omega(\{u, v\}) + 2\,\omega(\{v, v\}) & \{v, v\} \in E. \end{cases} \tag{1}$$

Then, the modularity, cf. [1], of a clustering $\mathcal{C}$ of G is defined by

$$\operatorname{mod}(\mathcal{C}) := \frac{\sum\limits_{C \in \mathcal{C}} \sum\limits_{\substack{\{u,v\} \in E \\ u,v \in C}} \omega(\{u, v\})}{\sum\limits_{e \in E} \omega(e)} \quad - \quad \frac{\sum\limits_{C \in \mathcal{C}} \left(\sum\limits_{v \in C} \zeta(v) \right)^2}{4 \left(\sum\limits_{e \in E} \omega(e) \right)^2}, \tag{2}$$

which is bounded by $-\frac{1}{2} \leq \operatorname{mod}(\mathcal{C}) \leq 1$ (see the appendix).

Finding a clustering $\mathcal{C}$ which maximises $\operatorname{mod}(\mathcal{C})$ is an NP-complete problem, i.e. ascertaining whether there exists a clustering that has at least a fixed modularity is strongly NP-complete [3, Thm. 4.4]. Hence, to find clusterings that have maximum modularity in reasonable time, we need to resort to heuristic algorithms. Many different clustering heuristics have been developed, for which we would like to refer the reader to the overview in [18, Sec. 5] and the references contained therein: there are heuristics based on spectral methods, maximum flow, graph bisection, betweenness, Markov chains, and random walks. The clustering method we present belongs to the category of bottom-up greedy agglomerative heuristics [2,14,16,21]. A massively parallel distributed-memory implementation of agglomerative clustering is provided in [17].

2 Clustering

We will now rewrite eq. (2) to a more convenient form. Let $C \in \mathcal{C}$ be a cluster and define the weight of a cluster as $\zeta(C) := \sum_{v \in C} \zeta(v)$, the set of all internal edges as $\operatorname{int}(C) := \{\{u, v\} \in E \mid u, v \in C\}$, the set of all external edges as $\operatorname{ext}(C) := \{\{u, v\} \in E \mid u \in C, v \notin C\}$, and for another cluster $C' \in \mathcal{C}$, the set of all cut edges between C and C' as $\operatorname{cut}(C, C') := \{\{u, v\} \in E \mid u \in C, v \in C'\}$. Let furthermore $\Omega := \sum_{e \in E} \omega(e)$ be the sum of all edge weights.

With these definitions, we can reformulate eq. (2) as (see the appendix):

$$\operatorname{mod}(\mathcal{C}) = \frac{1}{4\,\Omega^2} \sum_{C \in \mathcal{C}} \left[\zeta(C)\,(2\,\Omega - \zeta(C)) - 2\,\Omega \left(\sum_{\substack{C' \in \mathcal{C} \\ C' \neq C}} \omega(\operatorname{cut}(C, C')) \right) \right]. \qquad (3)$$

This way of looking at the modularity is useful for reformulating the agglomerative heuristic in terms of graph coarsening, as we will see in Sec. 2.1.

For this purpose, we also need to determine what effect the merging of two clusters has on the clustering's modularity. Let $\mathcal{C}$ be a clustering and $C, C' \in \mathcal{C}$. If we merge C and C' into one cluster $C \cup C'$, then the clustering $\mathcal{C}' := (\mathcal{C} \setminus \{C, C'\}) \cup \{C \cup C'\}$ we obtain, has modularity (see the appendix)

$$\operatorname{mod}(\mathcal{C}') = \operatorname{mod}(\mathcal{C}) + \frac{1}{2\,\Omega^2} \Big(2\,\Omega\,\omega(\operatorname{cut}(C, C')) - \zeta(C)\,\zeta(C') \Big), \qquad (4)$$

and the new cluster has weight

$$\zeta(C \cup C') = \sum_{v \in C} \zeta(v) + \sum_{v \in C'} \zeta(v) = \zeta(C) + \zeta(C'). \qquad (5)$$

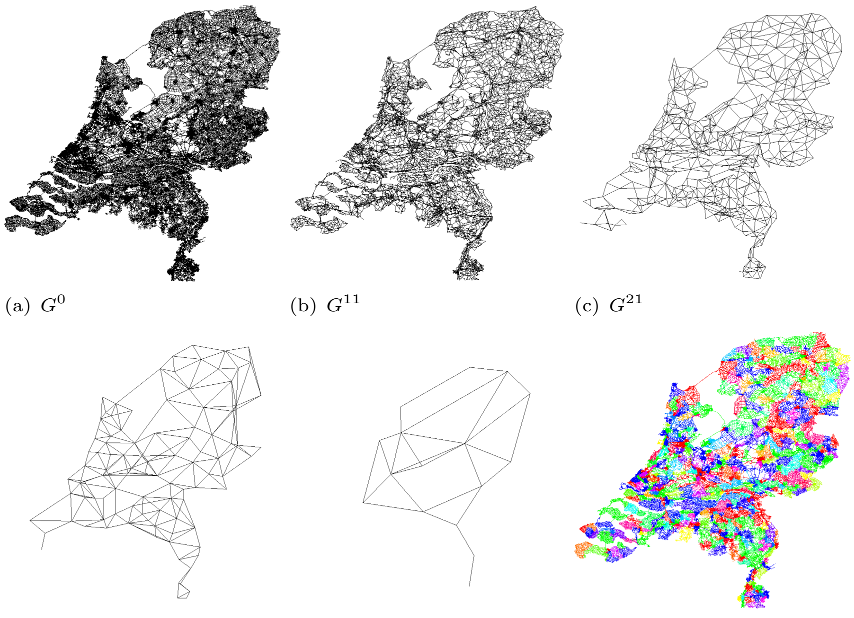
2.1 Agglomerative heuristic

Eq. (3), eq. (4), and eq. (5) suggest an agglomerative heuristic to generate a clustering [14,17,21]. Let $G = (V, E, \omega, \zeta)$ be a weighted undirected graph for which we want to calculate a clustering $\mathcal{C}$ of high modularity, provided with edge weights ω and vertex weights ζ as defined by eq. (1).

We start out with a clustering where each vertex of the original graph is a separate cluster, and then progressively merge these clusters to increase the modularity of the clustering. This process is illustrated in Fig. 1. The decision which pairs of clusters to merge is based on eq. (4): we generate a weighted matching in the graph with as vertices all the current clusters and as edges the sets $\{C, C'\}$ for which $\operatorname{cut}(C, C') \neq \emptyset$. The weight of such an edge $\{C, C'\}$ is then given by eq. (4), such that a maximum-weight matching will result in pairwise mergings of clusters for which the increase of the modularity is maximal.

We do this formally by, starting with G, constructing a sequence of weighted graphs $G^i = (V^i, E^i, \omega^i, \zeta^i)$ with surjective maps $\pi^i : V^i \to V^{i+1}$,

$$G \quad = \quad G^0 \quad \overset{\pi^0}{\to} \quad G^1 \quad \overset{\pi^1}{\to} \quad G^2 \quad \overset{\pi^2}{\to} \quad \ldots$$



(a) G^0 (b) G^{11} (c) G^{21}

(d) G^{26} (e) G^{33} (f) Best clustering (G^{21}).

Fig. 1. Clustering of `netherlands` into 506 clusters with modularity 0.995.

These graphs G^i correspond to clusterings $\mathcal{C}^i$ for G in the following way:

$$\mathcal{C}^i := \{\{v \in V \mid (\pi^{i-1} \circ \cdots \circ \pi^0)(v) = u\} \mid u \in V^i\}, \qquad i = 0, 1, 2, \ldots$$

Each vertex of the graph G^i will correspond to precisely one cluster in $\mathcal{C}^i$: all vertices of G that were merged together into a single vertex in G^i via π^0, ..., π^{i-1}, are considered as a single cluster. (In particular for $G^0 = G$ each vertex of the original graph is a separate cluster.)

From eq. (5) we know that weights $\zeta(\cdot)$ of merged clusters should be summed, while for calculating the modularity, eq. (3), and the change in modularity due to merging, eq. (4), we only need the total edge weight $\omega(\mathrm{cut}(\cdot,\cdot))$ of the collection of edges between two clusters, not of individual edges. Hence, when merging two clusters, we can safely merge the edges in G^i that are mapped to a single edge in G^{i+1} by π^i, provided we sum their edge weights. This means that the merging of clusters in G^i to obtain G^{i+1} corresponds precisely to coarsening the graph G^i to G^{i+1}. Furthermore, weighted matching in the graph of all current clusters corresponds to a weighted matching in G^i where we consider edges $\{u^i, v^i\} \in E^i$ to have weight $2\,\Omega\,\omega^i(\{u^i, v^i\}) - \zeta(u^i)\,\zeta(v^i)$ during matching. This entire procedure is outlined in Alg. 1, where we use a map $\mu : V \to \mathbf{N}$ to indicate matchings $M \subseteq E$ by letting $\mu(u) = \mu(v) \iff \{u, v\} \in M$ for vertices $u, v \in V$.

Algorithm 1 Agglomerative clustering heuristic for a weighted undirected graph $G = (V, E, \omega, \zeta)$ with ζ given by eq. (1). Produces a clustering $\mathcal{C}$ of G.

```
1: $\mathrm{mod}^{\mathrm{best}} \leftarrow -\infty$
2: $G^0 = (V^0, E^0, \omega^0, \zeta^0) \leftarrow G$
3: $i \leftarrow 0$
4: $\mathcal{C}^0 \leftarrow \{\{v\} \mid v \in V\}$
5: while $|V^i| > 1$ do
6:     if $\mathrm{mod}(G, \mathcal{C}^i) \geq \mathrm{mod}^{\mathrm{best}}$ then
7:         $\mathrm{mod}^{\mathrm{best}} \leftarrow \mathrm{mod}(G, \mathcal{C}^i)$
8:         $\mathcal{C}^{\mathrm{best}} \leftarrow \mathcal{C}^i$
9:     $\mu \leftarrow$ match_clusters$(G^i)$
10:    $(\pi^i, G^{i+1}) \leftarrow$ coarsen$(G^i, \mu)$
11:    $\mathcal{C}^{i+1} \leftarrow \{\{v \in V \mid (\pi^i \circ \cdots \circ \pi^0)(v) = u\} \mid u \in V^{i+1}\}$
12:    $i \leftarrow i + 1$
13: return $\mathcal{C}^{\mathrm{best}}$
```

3 Graph coarsening

Graph coarsening is the merging of vertices in a graph to obtain a coarser version of the graph. Doing this recursively, we obtain a sequence of increasingly coarser approximations of the original graph. Such a multilevel view of the graph is useful for graph partitioning [4,8,11], but can also be used for clustering [21].

Let $G = (V, E, \omega, \zeta)$ be an undirected graph with edge weights ω and vertex weights ζ. A *coarsening* of G is a map $\pi : V \to V'$ together with a graph $G' = (V', E', \omega', \zeta')$ satisfying the following properties:

1. $\pi(V) = V'$,
2. $\pi(E) = \{\{\pi(u), \pi(v)\} \mid \{u, v\} \in E\} = E'$,
3. for $v' \in V'$,
$$\zeta'(v') = \sum_{\substack{v \in V \\ \pi(v) = v'}} \zeta(v), \tag{6}$$
4. and for $e' \in E'$,
$$\omega'(e') = \sum_{\substack{\{u,v\} \in E \\ \{\pi(u), \pi(v)\} = e'}} \omega(\{u, v\}). \tag{7}$$

Let $\mu : V \to \mathbf{N}$ be a map indicating the desired coarsening, such that vertices u and v should be merged into a single vertex precisely when $\mu(u) = \mu(v)$. Then we call a coarsening π *compatible with* μ if $\pi(u) = \pi(v)$ if and only if $\mu(u) = \mu(v)$ for all $u, v \in V$. The task of the coarsening algorithm is, given G and μ, to generate a graph coarsening π, G' that is compatible with μ.

As noted at the end of Sec. 2.1, the map μ can correspond to a matching M, by letting $\mu(u) = \mu(v)$ if and only if the edge $\{u, v\} \in M$. This ensures that we do not coarsen the graph too aggressively, only permitting a vertex to be merged with at most one other vertex during coarsening. Such a coarsening approach is also used in hypergraph partitioning [19]. For our coarsening algorithm however, it is not required that μ is derived from a matching: any map $\mu : V \to \mathbf{N}$ is permitted.

3.1 Star-like graphs

The reason for permitting a general μ (i.e. where more than two vertices are contracted to a single vertex during coarsening), instead of a map μ arising from graph matchings is that the recursive coarsening process can get stuck on star-like graphs [5, Sec. 4.3].

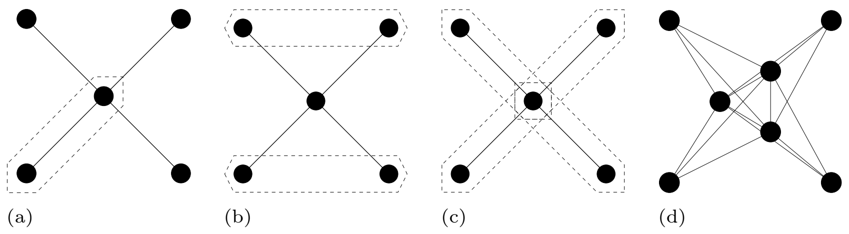


Fig. 2. Merging vertices in star-like graphs: by matching in (a), by merging vertices with the same neighbours in (b), and by merging more than two vertices in (c). In (d) we see a star-like graph with a centre clique of 3 vertices and 4 satellites.

In Fig. 2(a), we see a star graph in which a maximum matching is indicated. Coarsening this graph by matching the two matched vertices will yield a graph with only one vertex less. In general, with a k-pointed star, coarsening by matching will reduce the total number of vertices from $k+1$ to k, requiring k coarsening steps to reduce the star to a single vertex. This is slow compared to a graph for which we can find a perfect matching at each step of the coarsening, where the total number of vertices is halved at each step and we require only $\log_2(k)$ coarsening steps to reduce the graph to a single vertex. Hence, star graphs increase the number of coarsening iterations at line 5 of Alg. 1 we need to perform, which increases running time and has an adverse effect on parallelisability, because of the few matches that can actually be made in each iteration.

A way to remedy this problem is to identify vertices with the same neighbours and match these pairwise, see Fig. 2(b) [6,9]. When maximising clustering modularity however, this is not a good idea: for clusters $C, C' \in \mathcal{C}$ without any edges between them, $\text{cut}(C, C') = \emptyset$, merging C and C' will change the modularity by $\frac{-1}{2\,\Omega^2}\,\zeta(C)\,\zeta(C') \leq 0$.

Because of this, we will use the strategy from Fig. 2(c), and merge multiple outlying vertices, referred to as *satellites* from now on, to the centre of the star simultaneously. To do so, however, we need to be able to identify star centres and satellites in the graph.

As the defining characteristic of the centre of a star is its high degree, we will use the vertex degrees to measure to what extent a vertex is a centre or a satellite. We propose, for vertices $v \in V$, to let

$$\operatorname{cp}(v) := \frac{\deg(v)^2}{\sum\limits_{u \in V_v} \deg(u)}, \tag{8}$$

be the *centre potential* of v. Note that for satellites the centre potential will be small, because a satellite's degree is low, while the centre to which it is connected has a high degree. On the other hand, a star centre will have a high centre potential because of its high degree. Let us make this a little more precise.

For a regular graph where $\deg(v) = k$ for all $v \in V$, the centre potential will equal $\operatorname{cp}(v) = k^2/k^2 = 1$ for all vertices $v \in V$. Now consider a star-like graph, consisting of a clique of l vertices in the centre which are surrounded by k satellites that are connected to every vertex in the clique, but not to other satellites (Fig. 2(d) has $l = 3$ and $k = 4$), with $0 < l < k$. In such a graph, $\deg(v) = l$ for satellites v and $\deg(u) = l - 1 + k$ for vertices u in the centre clique. Hence, for satellites v

$$\operatorname{cp}(v) = \frac{l^2}{l\,(l - 1 + k)} \leq \frac{l}{l - 1 + l + 1} = \frac{1}{2},$$

while for centre vertices u

$$\operatorname{cp}(u) = \frac{(l - 1 + k)^2}{(l-1)\,(l - 1 + k) + k\,l} = 1 + \left(\frac{k - 1}{2\,l - 1 + \frac{(l-1)^2}{k}} \right) \geq \frac{4}{3}.$$

If we fix $l > 0$ and let the number of satellites $k \to \infty$, we see that

$$\operatorname{cp}(v) \to 0 \qquad \text{and} \qquad \operatorname{cp}(u) \to \infty.$$

Hence, the centre potential seems to be a good indicator for determining whether vertices v are satellites, $\operatorname{cp}(v) \leq \frac{1}{2}$, or centres, $\operatorname{cp}(v) \geq \frac{4}{3}$.

In Alg. 1, we will therefore, after line 9, use $\operatorname{cp}(v)$ to identify all satellites in the graph and merge these with the neighbouring non-satellite vertex that will yield the highest increase of modularity as indicated by eq. (4). This will both provide greedy modularity maximisation, and stop star-like graphs from slowing down the algorithm.

4 Parallel implementation

To make the description of the algorithm more explicit, we will need to deviate from some of the graph definitions of the introduction. First of all, we consider arrays in memory as ordered lists, and suppose that the vertices of the graph $G = (V, E, \omega, \zeta)$ to be coarsened are given by $V = (1, 2, \ldots, |V|)$. We index such lists with parenthesis, e.g. $V(2) = 2$, and denote their length by $|V|$. Instead of storing the edges E and edge weights ω of a graph explicitly, we will store for each vertex $v \in V$ the set of all its neighbours V_v, and include the edge weights

ω in this list. We will refer to these sets as *extended neighbour lists* and denote them by V_v^ω for $v \in V$.

Let us consider a small example: a graph with 3 vertices and edges $\{1, 2\}$ and $\{1, 3\}$ with edge weights $\omega(\{1, 2\}) = 4$ and $\omega(\{1, 3\}) = 5$. Then for the parallel coarsening algorithm we consider this graph as $V = (1, 2, 3)$, together with $V_1^\omega = ((2, 4), (3, 5))$ (since there are two edges originating from vertex 1, one going to vertex 2, and one going to vertex 3), $V_2^\omega = ((1, 4))$ (as $\omega(\{1, 2\}) = 4$), and $V_3^\omega = ((1, 5))$ (as $\omega(\{1, 3\}) = 5$).

In memory, such neighbour lists are stored as an array of indices and weights (in the small example, $((2, 4), (3, 5), (1, 4), (1, 5)))$, with for each vertex a range in this array (in the small example range $(1, 2)$ for vertex 1, $(3, 3)$ for 2, and $(4, 4)$ for 3). Note that we can extract all edges together with their weights ω directly from the extended neighbour lists. Hence, (V, E, ω, ζ) and $(V, \{V_v^\omega \mid v \in V\}, \zeta)$ are equivalent descriptions of G.

Algorithm 2 Parallel coarsening algorithm on the GPU, given a graph G with $V = (1, 2, \ldots, |V|)$ and a map $\mu : V \to \mathbf{N}$, this algorithm creates a graph coarsening π, G' compatible with μ.

```
 1: $\rho \leftarrow V$
 2: $(\rho, \mu) \leftarrow$ parallel_sort_by_key$(\rho, \mu)$
 3: $\mu \leftarrow$ parallel_adjacent_not_equal$(\mu)$
 4: $\pi^{-1} \leftarrow$ parallel_copy_index_if_nonzero$(\mu)$
 5: $V' \leftarrow (1, 2, \ldots, |\pi^{-1}|)$
 6: append$(\pi^{-1}, |V| + 1)$
 7: $\mu \leftarrow$ parallel_inclusive_scan$(\mu)$
 8: $\pi \leftarrow$ parallel_scatter$(\rho, \mu)$
 9: for $v' \in V'$ parallel do {Sum vertex weights.}
10:     $\zeta'(v') \leftarrow 0$
11:     for $i = \pi^{-1}(v')$ to $\pi^{-1}(v' + 1) - 1$ do
12:         $\zeta'(v') \leftarrow \zeta'(v') + \zeta(\rho(i))$
13: for $v' \in V'$ parallel do {Copy neighbours.}
14:     $V'^{\omega'}_{v'} \leftarrow \emptyset$
15:     for $i = \pi^{-1}(v')$ to $\pi^{-1}(v' + 1) - 1$ do
16:         for $(u, \omega) \in V^\omega_{\rho(i)}$ do
17:             append$(V'^{\omega'}_{v'}, (\pi(u), \omega))$
18: for $v' \in V'$ parallel do {Compress neighbours.}
19:     $V'^{\omega'}_{v'} \leftarrow$ compress_neighbours$(V'^{\omega'}_{v'})$
```

We will now discuss the parallel coarsening algorithm described by Alg. 2, in which the **parallel_*** functions are slight adaptations of those available in the Thrust template library [10]. The **for . . . parallel do** construct indicates a for-loop of which each iteration can be executed in parallel, independent of all other iterations.

We start with an undirected weighted graph G with vertices $V = (1, 2, \ldots, |V|)$, vertex weights ζ, and edges E with edge weights ω encoded in the extended neighbour lists as discussed above. A given map $\mu : V \to \mathbf{N}$ indicates which vertices should be merged to form the coarse graph.

Alg. 2 starts by creating an ordered list ρ of all the vertices V, and sorting ρ according to μ. The function **parallel_sort_by_key**(a, b) sorts both a and b such that $i \leq j \to b(a(i)) \leq b(a(j))$ for $1 \leq i, j \leq |a|$, and does so in parallel. Consider for example a graph with 12 vertices and a given μ:

ρ	1	2	3	4	5	6	7	8	9	10	11	12
μ	9	2	3	22	9	9	22	2	3	3	2	4

Then applying **parallel_sort_by_key** will yield

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	2	2	2	3	3	3	4	9	9	9	22	22

We then apply the function **parallel_adjacent_not_equal**(a) which sets $a(1)$ to 1, and for $1 < i \leq |a|$ sets $a(i)$ to 1 if $a(i) \neq a(i-1)$ and to 0 otherwise. This yields

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	0	0	1	0	0	1	1	0	0	1	0

Now we know where each group of vertices of G that needs to be merged together starts. We will store these numbers in the 'inverse' of the projection map π that we will construct later, such that we know, for each coarse vertex v', what vertices v in the original graph are coarsened to v'. The function **parallel_copy_index_if_nonzero**(a) picks out the indices $1 \leq i \leq |a|$ for which $a(i) \neq 0$ and stores these consecutively in a list, in parallel.

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	0	0	1	0	0	1	1	0	0	1	0
π^{-1}	1	4	7	8	11							

This gives us the number of vertices in the coarse graph as $|\pi^{-1}| = 5$, so $V' = (1, 2, \ldots, |\pi^{-1}|)$. To make sure we get a valid range for the last vertex in G', at line 6 we append $|V| + 1$ to π^{-1}. Now, we want to create the map $\pi : V \to V'$ relating the vertices of our original graph to the vertices of the coarse graph. We do this by re-enumerating μ using an inclusive scan. The function **parallel_inclusive_scan**(a) keeps a running sum s initialised as 0 and updates for $1 \leq i \leq |a|$ the value $s \leftarrow s + a(i)$, storing $a(i) \leftarrow s$.

ρ	2	8	11	3	9	10	12	1	5	6	4	7
μ	1	1	1	2	2	2	3	4	4	4	5	5
π^{-1}	1	4	7	8	11	13						

From these lists, we can see that vertices $3, 9, 10 \in V$ are mapped to the vertex $2 \in V'$ (so, we should have $\pi(3) = \pi(9) = \pi(10) = 2$), and from $2 \in V'$ we can recover $3, 9, 10 \in V$ by looking at values of ρ in the range $\pi^{-1}(2), \ldots, \pi^{-1}(2+1) - 1$. From the construction of ρ and μ we know that we should have that $\pi(\rho(i)) =$

$\mu(i)$ for our map $\pi : V \to V'$. Hence, we use the $c =$ **parallel_scatter**(a, b) function, which sets $c(a(i)) \leftarrow b(i)$ for $1 \leq i \leq |a|$ in parallel, to obtain π. Now we know both how to go from the original to the coarse graph (π), and from the coarse to the original graph (π^{-1} and ρ). This permits us to construct the extended neighbour lists of the coarse graph.

Let us look at this from the perspective of a single vertex $v' \in V'$ in the coarse graph. All vertices v in the fine graph that are mapped to v' by π are given by $\rho(\pi^{-1}(v')), \ldots, \rho(\pi^{-1}(v' + 1) - 1)$. All vertex weights (line 9) $\zeta(v)$ of these v are summed to satisfy eq. (6). By considering all extended neighbour lists V_v^{ω} (line 13), we can construct the extended neighbour list $V_{v'}'^{\omega'}$ of v'. Every element in the neighbour list is a pair $(u, \omega) \in V_v^{\omega}$. In the coarse graph, $\pi(u)$ will be a neighbour of v' in G', so we add $(\pi(u), \omega)$ to the extended neighbour list $V_{v'}'^{\omega'}$ of v'.

After copying all the neighbours, we compress the neighbour lists of each vertex in the coarse graph by first sorting elements $(u', \omega) \in V_{v'}'^{\omega'}$ of the extended neighbour list by u', and then merging ranges $((u', \omega_1), (u', \omega_2), \ldots, (u', \omega_k))$ in $V_{v'}'^{\omega'}$ to a single element $(u', \omega_1 + \omega_2 + \ldots + \omega_k)$ with **compress_neighbours**. This ensures that we satisfy eq. (7).

Afterwards, we have V', $\{V_{v'}'^{\omega'} \mid v' \in V'\}$, and ζ', together with a map $\pi : V \to V'$ compatible with the given μ.

4.1 Parallelisation of the remainder of Alg. 1

Now that we know how to coarsen the graph in parallel in Alg. 1 by using Alg. 2, we will also look at parallelising the other parts of the algorithm. We generate matchings μ on the GPU using the algorithm from [7], where we perform weighted matching with edge weight $2\,\Omega\,\omega(\{u, v\}) - \zeta(u)\,\zeta(v)$ (cf. eq. (4)), for each edge $\{u, v\} \in E$.

Satellites can be marked and merged in parallel as described by Alg. 3, where the matching algorithm indicates that a vertex has not been matched to any other vertex by using a special value for μ, such that the validity of $|\mu^{-1}(\{\mu(v)\})| = 1$ can be checked very quickly. Note that in this case the gain of merging a satellite with a non-satellite as described by eq. (4) is only an approximation, since we can merge several satellites simultaneously in parallel.

In Alg. 1 (line 11), we can also keep track of clusters in parallel. We create a clustering map $\kappa : V \to \mathbf{N}$ that indicates the cluster index of each vertex of the original graph, such that for $i = 0, 1, \ldots$, our clustering will be $\mathcal{C}^i = \{\{v \in V \mid \kappa^i(v) = k\} \mid k \in \mathbf{N}\}$ (i.e. vertices u and v belong to the same cluster precisely when $\kappa^i(u) = \kappa^i(v)$). Initially we assign all vertices to a different cluster by letting $\kappa^0(v) \leftarrow v$ for all $v \in V$. After coarsening, the clustering is then updated at line 11 by setting $\kappa^{i+1}(v) \leftarrow \pi^i(\kappa^i(v))$. We do this in parallel using $c \leftarrow$ **parallel_gather**(a, b), which sets $c(i) \leftarrow b(a(i))$ for $1 \leq i \leq |b|$.

Note that unlike [16,21], we do not employ a local refinement strategy such as Kernighan–Lin [12] to improve the quality of the obtained clustering from Alg. 1, because such an algorithm does not lend itself well to parallelisation. This is primarily caused by the fact that exchanging a single vertex between

Algorithm 3 Algorithm for marking and merging unmatched satellites in a given graph $G = (V, E, \omega, \zeta)$, extending a map $\mu : V \to \mathbf{N}$.

```
 1: for $v \in V$ parallel do {Mark unmatched satellites.}
 2:   if $|\mu^{-1}(\{\mu(v)\})| = 1$ and $\mathrm{cp}(v) \leq \frac{1}{2}$ then
 3:     $\sigma(v) \leftarrow$ true
 4:   else
 5:     $\sigma(v) \leftarrow$ false
 6: for $v \in V$ parallel do {Merge unmatched satellites.}
 7:   if $\sigma(v)$ then
 8:     $u^{\text{best}} \leftarrow \infty$
 9:     $w^{\text{best}} \leftarrow -\infty$
10:     for $u \in V_v$ do
11:       $w \leftarrow 2\,\Omega\,\omega(\{u, v\}) - \zeta(u)\,\zeta(v)$
12:       if $w > w^{\text{best}}$ and not $\sigma(u)$ then
13:         $w^{\text{best}} \leftarrow w$
14:         $u^{\text{best}} \leftarrow u$
15:     if $u^{\text{best}} \neq \infty$ then
16:       $\mu(v) \leftarrow \mu(u^{\text{best}})$
```

two clusters changes the total weight of both clusters, leading to a change in the modularity gain of *all* vertices in both the clusters. A parallel implementation of the Kernighan–Lin algorithm for clustering is therefore even more difficult than for graph partitioning [8,11], where exchanging vertices only affects the vertex's neighbours. Remedying this is an interesting avenue for further research.

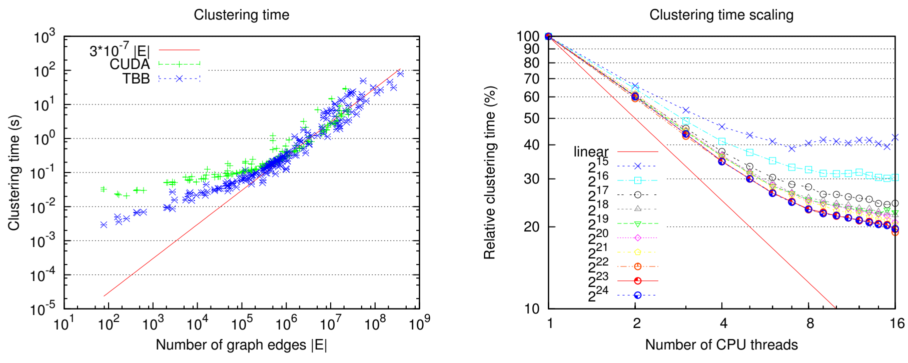
To further improve the performance of Alg. 1, we make use of two additional observations. We found during our clustering experiments that the modularity would first increase as the coarsening progressed and then would decrease after a peak value was obtained, as is also visible in [15, Fig. 6 and 9]. Hence, we stop Alg. 1 after the current modularity drops below 95% (to permit small fluctuations) of the highest modularity encountered thus far.

The second optimisation makes use of the fact that we do not perform uncoarsening steps in Alg. 1 (although with the data generated by Alg. 2 this is certainly possible), which makes it unnecessary to store the entire hierarchy G^0, G^1, G^2, ... in memory. Therefore, we only store two graphs, G^0 and G^1, and coarsen G^0 to G^1 as before, but then we coarsen G^1 to G^0, instead of a new graph G^2, and alternate between G^0 and G^1 as we coarsen the graph further.

5 Results

Alg. 1 was implemented using NVIDIA's Compute Unified Device Architecture (CUDA) language, together with the Thrust template library [10] on the GPU and using Intel's Threading Building Blocks (TBB) library on the CPU. The experiments were performed on a computer equipped with two quad-core 2.4 GHz Intel Xeon E5620 processors with hyperthreading (we use 16 threads), 24 GiB RAM, and an NVIDIA Tesla C2050 with 2687 MiB global memory. All source code for the algorithms, together with the scripts required to generate the benchmark data, have been released under the GNU General Public Licence and are freely available from `http://www.staff.science.uu.nl/~faggi101/`. It is important to note that the clustering times listed in Table 1 and Fig. 3 do include data transfer times from CPU to GPU, but not data transfer from hard disk to CPU memory. The recorded time and modularity are averaged over 16 runs, because of the use of random numbers in the matching algorithm [7]. These are generated using the TEA-4 algorithm [20] to improve performance.

The quality of the clusterings generated by the CPU implementation is generally a little higher (e.g. `eu-2005`) than those generated by the GPU, because the CPU version initially generates matchings consisting only of edges for which the incremental term in eq. (4) is non-negative. This is done as long as the coarsening sufficiently reduces the number of vertices, after which all edges are considered for matching, exactly once. The GPU version always considers all edges for matching, even if including them in the matching leads to a decrease of modularity.



(a) (b)

Fig. 3. In (a), we show the clustering time required by Alg. 1 for graphs from the 10th DIMACS challenge [1] test set (categories `clustering/`, `streets/`, `coauthor/`, `kronecker/`, `matrix/`, `random/`, `delaunay/`, `walshaw/`, `dyn-frames/`, and `redistrict/`), for both the CUDA and TBB implementations and show that, for large graphs, clustering time scales almost linearly with the number of edges. In (b), we show the parallel scaling of the TBB implementation of Alg. 1 as a function of the number of threads, normalised to the time required by a single-threaded run for graphs `rgg_n_2_`k`_s0` with 2^k vertices, from the `random/` category. We compare this to ideal, linear, scaling. The test system has 8 cores and up to 16 threads with hyperthreading.

Comparing Table 1 with modularities from [16, Table 1] for `karate` (0.412), `jazz` (0.444), `email` (0.572), and `PGPgiantcompo` (0.880), we see that Alg. 1 generates clusterings of lesser modularity. We attribute this to the absence of a local refinement strategy in Alg. 1, as noted in Sec. 4.1. The quality of the

G	$\lvert V\rvert$	$\lvert E\rvert$	mod_1	t_1	mod_2	t_2
`karate`	34	78	0.363	0.032	0.383	0.003
`jazz`	198	2,742	0.314	0.045	0.369	0.009
`email`	1,133	5,451	0.440	0.073	0.473	0.016
`PGPgiantcompo`	10,680	24,316	0.809	0.088	0.841	0.029
`cond-mat`	16,726	47,594	0.788	0.113	0.798	0.053
`as-22july06`	22,963	48,436	0.607	0.182	0.630	0.029
`cond-mat-2003`	31,163	120,029	0.674	0.190	0.689	0.072
`astro-ph`	16,706	121,251	0.588	0.214	0.610	0.059
`cond-mat-2005`	40,421	175,691	0.624	0.245	0.640	0.085
`preferentialAttachment`	100,000	499,985	0.214	1.208	0.216	0.211
`smallworld`	100,000	499,998	0.636	0.471	0.663	0.157
`G_n_pin_pout`	100,000	501,198	0.241	0.868	0.247	0.215
`caidaRouterLevel`	192,244	609,066	0.768	0.510	0.791	0.180
`cnr-2000`	325,557	2,738,969	0.828	2.126	0.904	0.376
`in-2004`	1,382,908	13,591,473	0.946	4.460	0.974	1.849
`eu-2005`	862,664	16,138,468	0.816	8.992	0.889	1.950
`road_central`	14,081,816	16,933,413	0.996	4.566	0.996	13.738
`road_usa`	23,947,347	28,854,312	-	-.-	0.997	21.695
`uk-2002`	18,520,486	261,787,258	-	-.-	0.974	31.008
`luxembourg.osm`	114,599	119,666	0.986	0.112	0.987	0.117
`belgium.osm`	1,441,295	1,549,970	0.992	0.433	0.993	1.136
`netherlands.osm`	2,216,688	2,441,238	0.994	0.591	0.995	1.791
`italy.osm`	6,686,493	7,013,978	0.997	1.530	0.997	5.637
`great-britain.osm`	7,733,822	8,156,517	0.997	1.810	0.997	6.267
`germany.osm`	11,548,845	12,369,181	0.997	2.815	0.997	10.096
`asia.osm`	11,950,757	12,711,603	0.998	2.655	0.998	10.332
`europe.osm`	50,912,018	54,054,660	-	-.-	0.999	49.087
`coAuthorsCiteseer`	227,320	814,134	0.837	0.417	0.847	0.211
`coAuthorsDBLP`	299,067	977,676	0.748	0.598	0.760	0.270
`citationCiteseer`	268,495	1,156,647	0.643	0.909	0.683	0.307
`coPapersDBLP`	540,486	15,245,729	0.640	6.565	0.667	2.337
`coPapersCiteseer`	434,102	16,036,720	0.746	6.595	0.775	2.347
`kron_g500-simple-logn16`	65,536	2,456,071	0.030	3.191	0.031	0.504
`kron_g500-simple-logn17`	131,072	5,113,985	0.027	6.974	0.028	1.220
`kron_g500-simple-logn18`	262,144	10,582,686	0.025	14.714	0.025	2.808
`kron_g500-simple-logn19`	524,288	21,780,787	0.023	28.904	0.023	6.442
`kron_g500-simple-logn20`	1,048,576	44,619,402	-	-.-	0.022	14.528
`kron_g500-simple-logn21`	2,097,152	91,040,932	-	-.-	0.020	31.584
`ldoor`	952,203	22,785,136	0.945	6.772	0.949	3.019
`audikw1`	943,695	38,354,076	-	-.-	0.858	5.163
`cage15`	5,154,859	47,022,346	-	-.-	0.680	13.872

Table 1. For graphs $G = (V, E)$, this table lists the average modularities $\mathrm{mod}_{1,2}$, eq. (2), of clusterings of G generated in an average time of $t_{1,2}$ seconds by the CUDA_1 and TBB_2 implementations of Alg. 1. A '-' indicates that the GPU ran out of memory. Results are averaged over 16 runs. Top to bottom, this table lists graphs from the `clustering/`, `streets/`, `coauthor/`, `kronecker/`, and `matrix/` categories of the 10th DIMACS challenge [1].

clusterings of irregular graphs from the `kronecker/` categories is an order of magnitude smaller than those of graphs from other categories. We are uncertain about what causes this behaviour.

Alg. 1 is fast: for the `road_central` graph with 14 million vertices and 17 million edges, the GPU generates a clustering with modularity 0.996 in 4.6 seconds, while for `uk-2002`, with 18 million vertices and 262 million edges, the CPU generates a clustering with modularity 0.974 in 31 seconds. In particular, for clustering of nearly regular graphs (i.e. where the ratio $\big(\max_{v \in V} \deg(v)\big)/\big(\min_{u \in V} \deg(u)\big)$ is small) such as street networks, the high bandwidth of the GPU enables us to find high-quality clusterings in very little time (Table 1). Furthermore, Fig. 3(a) suggests that in practice, Alg. 1 scales linearly with the number of edges of the graph, while Fig. 3(b) shows that the parallel performance of the algorithm scales reasonably with the number of available cores, increasingly so as the size of the graph increases. Note that with dual quad-core processors, we have eight physical cores available, which explains the smaller increase in performance when the number of threads is extended beyond eight via hyperthreading.

From Fig. 3(a), we see that while the GPU performs well for large, $|E| \geq 10^6$, nearly regular graphs, the CPU handles small and irregular graphs better. This can be explained by the GPU setup time and CPU to GPU data transfer time that become dominant for small graphs, and by the fact that for large irregular graphs, vertices with a higher-than-average degree keep one of the threads occupied, while the threads treating the other, low-degree, vertices are already done, leading to a low GPU occupancy.

6 Conclusion

In this paper we have presented a fine-grained shared-memory parallel algorithm for graph coarsening, Alg. 2, suitable for both multi-core CPUs and GPUs. Through a greedy agglomerative clustering heuristic, Alg. 1, we try to find graph clusterings of high modularity to measure the performance of this coarsening method. Our parallel clustering algorithm scales well for large graphs if the number of threads is increased, Fig. 3(b), and can generate clusterings of reasonable quality in very little time, requiring 4.6 seconds to generate a modularity 0.996 clustering of a graph with 14 million vertices and 17 million edges.

An interesting direction for future research would be the development of a local refinement method for clustering, that scales well with the number of available processing cores, and can be implemented efficiently on GPUs. This would greatly benefit the quality of the generated clusterings.

7 Acknowledgements

We would like to thank Fredrik Manne for his insights in parallel matching and coarsening, and the Little Green Machine project, `http://littlegreenmachine.org/`, for permitting us to use their hardware under project NWO-M 612.071.305.

References

1. Bader, D.A., Sanders, P., Wagner, D., Meyerhenke, H., Hendrickson, B., Johnson, D.S., Walshaw, C., Mattson, T.G.: 10th DIMACS implementation challenge - graph partitioning and graph clustering (2012), `http://www.cc.gatech.edu/dimacs10/index.shtml`
2. Bisgin, H., Agarwal, N., Xu, X.: Does similarity breed connection? - an investigation in Blogcatalog and Last.fm communities. In: Proc of. SocialCom/PASSAT'10. pp. 570–575 (2010)
3. Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., Wagner, D.: On modularity clustering. IEEE Trans. Knowledge and Data Engineering 20(2), 172–188 (2008)
4. Bui, T., Jones, C.: A heuristic for reducing fill-in in sparse matrix factorization. In: Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing. pp. 445–452. SIAM, Philadelphia, PA, USA (1993)
5. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM TOMS 38(1), 1:1–1:25 (2011)
6. Duff, I.S., Reid, J.K.: Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. ACM TOMS 22, 227–257 (1996)
7. Fagginger Auer, B.O., Bisseling, R.H.: A GPU algorithm for greedy graph matching. In: Proc. FMC '11 (to appear). LNCS, Springer (2011)
8. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Proc. Supercomputing '95. ACM, New York, NY, USA (1995)
9. Hendrickson, B., Rothberg, E.: Improving the run time and quality of nested dissection ordering. SIAM J. Sci. Comput. 20(2), 468–489 (1998)
10. Hoberock, J., Bell, N.: Thrust: A parallel template library (2010), `http://www.meganewtons.com/`, version 1.3.0
11. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: Proc. Supercomputing '95. p. 29. ACM, New York, NY, USA (1995)
12. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal 49, 291–307 (1970)
13. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Statistical properties of community structure in large social and information networks. In: Proc. WWW '08. pp. 695–704. ACM, New York, NY, USA (2008)
14. Newman, M.E.J.: Fast algorithm for detecting community structure in networks. Phys. Rev. E 69, 066133 (2004)
15. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113 (2004)
16. Ovelgönne, M., Geyer-Schulz, A., Stein, M.: Randomized greedy modularity optimization for group detection in huge social networks. In: Proc. SNA-KDD '10. ACM, Washington, DC, USA (2010)
17. Riedy, E.J., Meyerhenke, H., Ediger, D., Bader, D.A.: Parallel community detection for massive graphs. In: Proc. PPAM11. Springer, Torun, Poland (2011)
18. Schaeffer, S.E.: Graph clustering. Computer Science Review 1(1), 27–64 (2007)
19. Vastenhouw, B., Bisseling, R.H.: A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. 47(1), 67–95 (2005)
20. Zafar, F., Olano, M., Curtis, A.: GPU random numbers via the tiny encryption algorithm. In: Proc. HPG10. pp. 133–141. Eurographics Association, Saarbrucken, Germany (2010)
21. Zhu, Z., Wang, C., Ma, L., Pan, Y., Ding, Z.: Scalable community discovery of large networks. In: Proc. WAIM '08. pp. 381–388 (2008)

8 Appendix

8.1 Reformulating modularity

Our first observation is that for every cluster $C \in \mathcal{C}$, by eq. (1):

$$\zeta(C) = 2\,\omega(\mathrm{int}(C)) + \omega(\mathrm{ext}(C)). \tag{9}$$

Now we rewrite eq. (2) using the definitions we gave before:

$$\begin{aligned}
\mathrm{mod}(\mathcal{C}) &= \frac{\sum_{C\in\mathcal{C}} \omega(\mathrm{int}(C))}{\varOmega} - \frac{\sum_{C\in\mathcal{C}} \zeta(C)^2}{4\,\varOmega^2} \\
&= \frac{1}{4\,\varOmega^2} \sum_{C\in\mathcal{C}} \left(4\,\varOmega\,\omega(\mathrm{int}(C)) - \zeta(C)^2\right) \\
&\overset{(9)}{=} \frac{1}{4\,\varOmega^2} \sum_{C\in\mathcal{C}} \left(4\,\varOmega\left[\frac{1}{2}\zeta(C) - \frac{1}{2}\omega(\mathrm{ext}(C))\right] - \zeta(C)^2\right).
\end{aligned}$$

Therefore, we arrive at the following expression,

$$\mathrm{mod}(\mathcal{C}) = \frac{1}{4\,\varOmega^2} \sum_{C\in\mathcal{C}} \Big(\zeta(C)\,(2\,\varOmega - \zeta(C)) - 2\,\varOmega\,\omega(\mathrm{ext}(C))\Big). \tag{10}$$

As

$$\mathrm{ext}(C) = \{\{u,v\} \in E \mid u \in C, v \notin C\} = \bigcup_{\substack{C'\in\mathcal{C}\\ C'\neq C}} \mathrm{cut}(C,C'),$$

as a disjoint union, we find eq. (3).

8.2 Merging clusters

Let $C, C' \in \mathcal{C}$ be a pair of different clusters, set $C'' = C \cup C'$ and let $\mathcal{C}' := (\mathcal{C} \setminus \{C, C'\}) \cup \{C''\}$ be the clustering obtained by merging C and C'.

Then $\zeta(C'') = \zeta(C) + \zeta(C')$ by eq. (5). Furthermore, as $\mathrm{cut}(C,C') = \mathrm{ext}(C) \cap \mathrm{ext}(C')$, we have that

$$\omega(\mathrm{ext}(C'')) = \omega(\mathrm{ext}(C)) + \omega(\mathrm{ext}(C')) - 2\,\omega(\mathrm{cut}(C,C')). \tag{11}$$

Using this, together with eq. (10), we find that

$$\begin{aligned}
4\,\varOmega^2(\mathrm{mod}(\mathcal{C}') - \mathrm{mod}(\mathcal{C})) &= -\zeta(C)\,(2\,\varOmega - \zeta(C)) + 2\,\varOmega\,\omega(\mathrm{ext}(C)) \\
&\quad - \zeta(C')\,(2\,\varOmega - \zeta(C')) + 2\,\varOmega\,\omega(\mathrm{ext}(C')) \\
&\quad + \zeta(C'')\,(2\,\varOmega - \zeta(C'')) - 2\,\varOmega\,\omega(\mathrm{ext}(C'')) \\
&\overset{(11)}{=} -\zeta(C)\,(2\,\varOmega - \zeta(C)) + 2\,\varOmega\,\omega(\mathrm{ext}(C)) \\
&\quad - \zeta(C')\,(2\,\varOmega - \zeta(C')) + 2\,\varOmega\,\omega(\mathrm{ext}(C')) \\
&\quad + (\zeta(C) + \zeta(C'))\,(2\,\varOmega - (\zeta(C) + \zeta(C'))) \\
&\quad - 2\,\varOmega\left[\omega(\mathrm{ext}(C)) + \omega(\mathrm{ext}(C')) - 2\,\omega(\mathrm{cut}(C,C'))\right] \\
&= 4\,\varOmega\,\omega(\mathrm{cut}(C,C')) - 2\,\zeta(C)\,\zeta(C').
\end{aligned}$$

So merging clusters C and C' from $\mathcal{C}$ to obtain a clustering $\mathcal{C}'$, leads to a change in modularity given by eq. (4).

8.3 Proof of $-\frac{1}{2} \leq \mathrm{mod}(\mathcal{C}) \leq 1$

This is shown in [3, Lem. 3.1] for the unweighted case. From eq. (2),

$$\mathrm{mod}(\mathcal{C}) \leq \frac{\sum\limits_{C\in\mathcal{C}} \sum\limits_{\substack{\{u,v\}\in E\\ u,v\in C}} \omega(\{u,v\})}{\sum\limits_{e\in E} \omega(e)} \quad - \quad 0 \leq \frac{\sum\limits_{\substack{\{u,v\}\in E\\ u,v\in V}} \omega(\{u,v\})}{\sum\limits_{e\in E} \omega(e)} = 1,$$

which shows one of the inequalities. For the other inequality, note that for any $C \in \mathcal{C}$ we have $0 \leq \omega(\mathrm{int}(C)) \leq \Omega - \omega(\mathrm{ext}(C))$, and therefore

$$\begin{aligned}
\mathrm{mod}(\mathcal{C}) &= \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(4\,\Omega\,\omega(\mathrm{int}(C)) - \zeta(C)^2\Big) \\
&\overset{(9)}{=} \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(4\,\Omega\,\omega(\mathrm{int}(C)) - 4\,\omega(\mathrm{int}(C))^2 - 4\,\omega(\mathrm{int}(C))\,\omega(\mathrm{ext}(C)) \\
&\qquad\qquad - \omega(\mathrm{ext}(C))^2\Big) \\
&= \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(4\,\omega(\mathrm{int}(C))\,[\Omega - \omega(\mathrm{ext}(C)) - \omega(\mathrm{int}(C))] - \omega(\mathrm{ext}(C))^2\Big) \\
&\geq \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(0 - \omega(\mathrm{ext}(C))^2\Big) \quad = -\sum_{C\in\mathcal{C}} \left(\frac{\omega(\mathrm{ext}(C))}{2\,\Omega}\right)^2.
\end{aligned}$$

Enumerate $\mathcal{C} = \{C_1, \ldots, C_k\}$ and define $x_i := \frac{\omega(\mathrm{ext}(C_i))}{2\,\Omega}$ for $1 \leq i \leq k$ to obtain a vector $x \in \mathbf{R}^k$. Note that $0 \leq x_i \leq \frac{1}{2}$ (as $0 \leq \omega(\mathrm{ext}(C_i)) \leq \Omega$) for $1 \leq i \leq k$, and because every external edge connects precisely two clusters, we have $\sum_{i=1}^k \omega(\mathrm{ext}(C_i)) \leq 2\,\Omega$, so $\sum_{i=1}^k x_i \leq 1$. By the above, we know that

$$\mathrm{mod}(\mathcal{C}) \geq -\|x\|_2^2,$$

hence we need to find an upper bound on $\|x\|_2^2$, for $x \in [0, \frac{1}{2}]^k$ satisfying $\sum_{i=1}^k x_i \leq 1$. For all $k \geq 2$, this upper bound equals $\|(\frac{1}{2}, \frac{1}{2}, 0, \ldots, 0)\|_2^2 = \frac{1}{2}$, so $\mathrm{mod}(\mathcal{C}) \geq -\frac{1}{2}$. The proof is completed by noting that for a single cluster, $\mathrm{mod}(\{V\}) = 0 \geq -\frac{1}{2}$.