Contemporary Mathematics

Graph Coarsening and Clustering on the GPU

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

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 [**8**], 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, 9, 12**] and multi-level clustering [**22**] and therefore forms a logical continuation of the research done in [**8**].

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 Section 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

2010 *Mathematics Subject Classification.* Primary 68R10, 68W10; Secondary 91C20, 05C70.
Key words and phrases. Graphs, GPU, shared-memory parallel, clustering.
This research was performed on hardware from NWO project NWO-M 612.071.305.

vertices $v \in V$, we denote the set of all of v's *neighbours* by

$$V_v := \{u \in V \mid \{u, v\} \in E\} \setminus \{v\}.$$

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, 14**]. 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 [**16**], 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

$$\zeta(v) := \begin{cases} \sum\limits_{\{u,v\} \in E} \omega(\{u, v\}) & \text{if } \{v, v\} \notin E, \\ \sum\limits_{\substack{\{u,v\} \in E \\ u \neq v}} \omega(\{u, v\}) + 2\,\omega(\{v, v\}) & \text{if } \{v, v\} \in E. \end{cases} \tag{1.1}$$

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

$$\operatorname{mod}(\mathcal{C}) := \frac{\sum\limits_{C \in \mathcal{C}} \sum\limits_{\substack{\{u,v\} \in E \\ u,v \in C}} \omega(\{u, v\})}{\sum\limits_{e \in E} \omega(e)} - \frac{\sum\limits_{C \in \mathcal{C}} \left(\sum\limits_{v \in C} \zeta(v) \right)^2}{4 \left(\sum\limits_{e \in E} \omega(e) \right)^2}, \tag{1.2}$$

which is bounded by $-\frac{1}{2} \leq \operatorname{mod}(\mathcal{C}) \leq 1$, as we show in 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**, Theorem 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 [**19**, Section 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 greedy agglomerative heuristics [**2, 5, 15, 17, 22**]. Our overall approach is similar to the parallel clustering algorithm discussed by Riedy et al. in [**18**] and a detailed comparison is included in Section 5.

2. Clustering

We will now rewrite (1.2) 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 $\text{int}(C) := \{\{u, v\} \in E \mid u, v \in C\}$, the set of all external edges as $\text{ext}(C) := \{\{u, v\} \in E \mid u \in C, v \notin C\}$, and for another cluster $C' \in \mathcal{C}$, the set of all cut edges between C and C' as $\text{cut}(C, C') := \{\{u, v\} \in E \mid u \in C, v \in C'\}$. Let furthermore $\Omega := \sum_{e \in E} \omega(e)$ be the sum of all edge weights.

With these definitions, we can reformulate (1.2) as (see the appendix):

$$\text{mod}(\mathcal{C}) = \frac{1}{4\,\Omega^2} \sum_{C \in \mathcal{C}} \left[\zeta(C)\,(2\,\Omega - \zeta(C)) - 2\,\Omega \left(\sum_{\substack{C' \in \mathcal{C} \\ C' \neq C}} \omega(\text{cut}(C, C')) \right) \right]. \tag{2.1}$$

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

$$\text{mod}(\mathcal{C}') = \text{mod}(\mathcal{C}) + \frac{1}{2\,\Omega^2} \Big(2\,\Omega\,\omega(\text{cut}(C, C')) - \zeta(C)\,\zeta(C') \Big), \tag{2.2}$$

and the new cluster has weight

$$\zeta(C \cup C') = \sum_{v \in C} \zeta(v) + \sum_{v \in C'} \zeta(v) = \zeta(C) + \zeta(C'). \tag{2.3}$$

2.1. Agglomerative heuristic. Equations (2.1), (2.2), and (2.3) suggest an agglomerative heuristic to generate a clustering [**15, 18, 22**]. Let $G = (V, E, \omega, \zeta)$ be a weighted undirected graph, provided with edge weights ω and vertex weights ζ as defined by (1.1), for which we want to calculate a clustering $\mathcal{C}$ of high modularity.

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 Figure 1. The decision which pairs of clusters to merge is based on (2.2): we generate a weighted matching in the graph with as vertices all the current clusters and the sets $\{C, C'\}$ for which $\text{cut}(C, C') \neq \emptyset$ as edges. The weight of such an edge $\{C, C'\}$ is then given by (2.2), such that a maximum-weight matching will result in pairwise mergings of clusters for which the increase of the modularity is maximal.

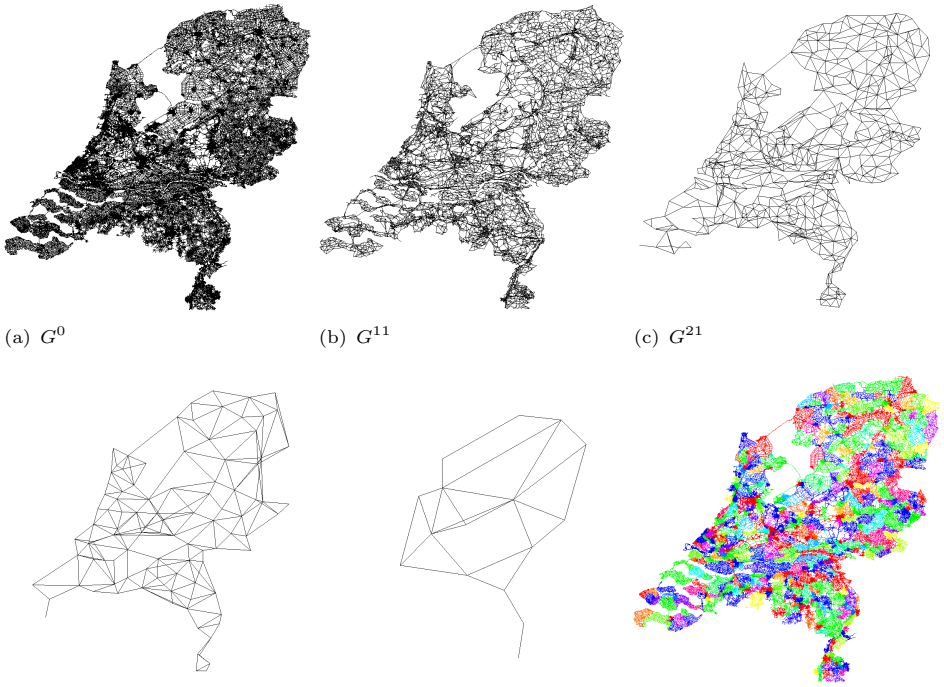
We do this formally by, starting with G, constructing a sequence of weighted graphs $G^i = (V^i, E^i, \omega^i, \zeta^i)$ with surjective maps $\pi^i : V^i \to V^{i+1}$,

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

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

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

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



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

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

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

From (2.3) we know that weights $\zeta(\cdot)$ of merged clusters should be summed, while for calculating the modularity, (2.1), and the change in modularity due to merging, (2.2), 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 $2\,\Omega\,\omega^i(\{u^i, v^i\}) - \zeta^i(u^i)\,\zeta^i(v^i)$ during matching. This entire procedure is outlined in Algorithm 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 $u, v \in V$.

3. 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, 9, 12**], but can also be used for clustering [**22**].

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:

Algorithm 1 Agglomerative clustering heuristic for a weighted undirected graph $G = (V, E, \omega, \zeta)$ with ζ given by (1.1). Produces a clustering $\mathcal{C}$ of G.

$\mathrm{mod}^{\mathrm{best}} \leftarrow -\infty$
$G^0 = (V^0, E^0, \omega^0, \zeta^0) \leftarrow G$
$i \leftarrow 0$
$\mathcal{C}^0 \leftarrow \{\{v\} \mid v \in V\}$
while $|V^i| > 1$ **do**
  if $\mathrm{mod}(G, \mathcal{C}^i) \geq \mathrm{mod}^{\mathrm{best}}$ **then**
    $\mathrm{mod}^{\mathrm{best}} \leftarrow \mathrm{mod}(G, \mathcal{C}^i)$
    $\mathcal{C}^{\mathrm{best}} \leftarrow \mathcal{C}^i$
  $\mu \leftarrow$ **match_clusters**(G^i)
  $(\pi^i, G^{i+1}) \leftarrow$ **coarsen**(G^i, μ)
  $\mathcal{C}^{i+1} \leftarrow \{\{v \in V \mid (\pi^i \circ \cdots \circ \pi^0)(v) = u\} \mid u \in V^{i+1}\}$
  $i \leftarrow i + 1$
return $\mathcal{C}^{\mathrm{best}}$

(1) $\pi(V) = V'$,
(2) $\pi(E) = \{\{\pi(u), \pi(v)\} \mid \{u, v\} \in E\} = E'$,
(3) for $v' \in V'$,

$$\zeta'(v') = \sum_{\substack{v \in V \\ \pi(v) = v'}} \zeta(v), \tag{3.1}$$

(4) and for $e' \in E'$,

$$\omega'(e') = \sum_{\substack{\{u,v\} \in E \\ \{\pi(u), \pi(v)\} = e'}} \omega(\{u, v\}). \tag{3.2}$$

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 for all $u, v \in V$ it holds that $\pi(u) = \pi(v)$ if and only if $\mu(u) = \mu(v)$. 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 Section 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 [**20**]. 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 [**6**, Section 4.3].

In Figure 2(a), we see a star graph in which a maximum matching is indicated. Coarsening this graph by merging 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

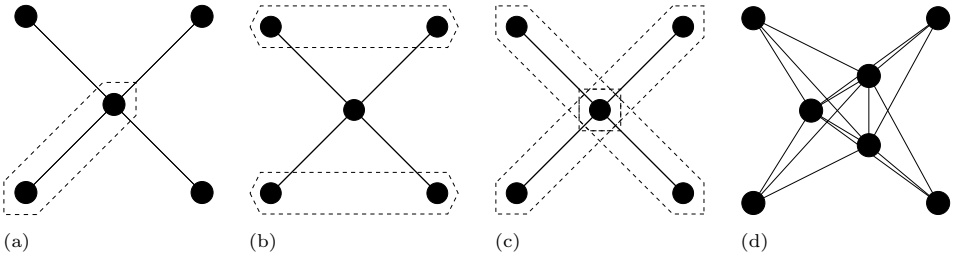


FIGURE 2. Merging vertices in star-like graphs: by matching in (a), by merging vertices with the same neighbours in (b), and by merging more than two vertices in (c). In (d) we see a star-like graph with a centre clique of 3 vertices and 4 satellites.

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 Algorithm 1 we need to perform, which increases running time and has an adverse effect on parallelisation, 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 Figure 2(b) [**7, 10**]. When maximising clustering modularity however, this is not a good idea: for clusters $C, C' \in \mathcal{C}$ without any edges between them, $\operatorname{cut}(C, C') = \emptyset$, 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 Figure 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

$$\operatorname{cp}(v) := \frac{\deg(v)^2}{\sum_{u \in V_v} \deg(u)}, \tag{3.3}$$

be the *centre potential* of v. Here, the *degree* of a vertex $v \in V$ is defined as $\deg(v) := |V_v|$. 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 (Figure 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

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

while for centre vertices u

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

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

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

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

In Algorithm 1, we will therefore, after line 9, use $\mathrm{cp}(v)$ 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 (2.2). This will both provide greedy modularity maximisation, and stop star-like graphs from slowing down the algorithm.

4. Parallel implementation

In this section, we will demonstrate how the different parts of the clustering algorithm can be implemented in a style that is suitable for the GPU.

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 parentheses, 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.

We will now discuss the parallel coarsening algorithm described by Algorithm 2, in which the **parallel_*** functions are slight adaptations of those available in the Thrust template library [**11**]. The **for . . . parallel do** construct indicates a for-loop of which each iteration can be executed in parallel, independent of all other iterations.

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

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.

Algorithm 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, 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, π^{-1} in this case, 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 $\pi : V \to V'$. Note that $\rho(i)$ is the original vertex in V and $\mu(i)$ is the current vertex in 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 (3.1). By considering all extended neighbour lists V_v^ω (line 13), we can construct the extended neighbour list $V_{v'}^{\prime\omega'}$ 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 $V_{v'}^{\prime\omega'}$ of v'.

After copying all the neighbours, we compress the neighbour lists of each vertex in the coarse graph by first sorting elements $(u', \omega) \in V_{v'}^{\prime\omega'}$ of the extended neighbour list by u', and then merging ranges $((u', \omega_1), (u', \omega_2), \ldots, (u', \omega_k))$ in $V_{v'}^{\prime\omega'}$ to a single element $(u', \omega_1 + \omega_2 + \ldots + \omega_k)$ with **compress_neighbours**. This ensures that we satisfy (3.2).

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

4.1. Parallelisation of the remainder of Algorithm 1. Now that we know how to coarsen the graph in parallel in Algorithm 1 by using Algorithm 2, we will also look at parallelising the other parts of the algorithm. We generate matchings μ on the GPU using the algorithm from [**8**], where we perform weighted matching with edge weight $2\,\Omega\,\omega(\{u, v\}) - \zeta(u)\,\zeta(v)$ (cf. (2.2)), for each edge $\{u, v\} \in E$.

Satellites can be marked and merged in parallel as described by Algorithm 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

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: $u^{\mathrm{best}} \leftarrow \infty$
9: $w^{\mathrm{best}} \leftarrow -\infty$
10: **for** $u \in V_v$ **do**
11: $w \leftarrow 2\,\Omega\,\omega(\{u, v\}) - \zeta(u)\,\zeta(v)$
12: **if** $w > w^{\mathrm{best}}$ **and not** $\sigma(u)$ **then**
13: $w^{\mathrm{best}} \leftarrow w$
14: $u^{\mathrm{best}} \leftarrow u$
15: **if** $u^{\mathrm{best}} \neq \infty$ **then**
16: $\mu(v) \leftarrow \mu(u^{\mathrm{best}})$

with a non-satellite as described by (2.2) is only an approximation, since we can merge several satellites simultaneously in parallel.

In Algorithm 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 $1 \leq i \leq |a|$.

Note that unlike [**17, 22**], we do not employ a local refinement strategy such as Kernighan–Lin [**13**] to improve the quality of the obtained clustering from Algorithm 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 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 [**9, 12**], where exchanging vertices only affects the vertex's neighbours. Remedying this is an interesting avenue for further research.

To further improve the performance of Algorithm 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 [**16**, Figure 6 and 9]. Hence, we stop Algorithm 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 Algorithm 1 (although with the data generated by Algorithm 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.

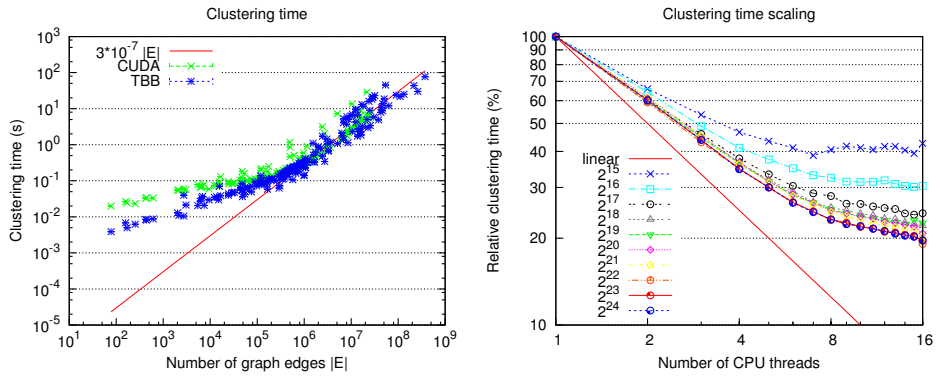
5. Results

Algorithm 1 was implemented using NVIDIA's Compute Unified Device Architecture (CUDA) language together with the Thrust template library [**11**] 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 C2075 with 5375 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 `https://github.com/BasFaggingerAuer/Multicore-Clustering`. It is important to note that the clustering times listed in Table 1, 2, and Figure 3 do include data transfer times from CPU to GPU, but not data transfer from hard disk to CPU memory. On average, 5.5% of the total running time is spent on CPU–GPU data transfer. The recorded time and modularity are averaged over 16 runs, because of the use of random numbers in the matching algorithm [**8**]. These are generated using the TEA-4 algorithm [**21**] to improve performance.

The modularity of the clusterings generated by the CPU implementation is generally a little higher (e.g. `eu-2005`) than those generated by the GPU. The difference between both algorithms is caused by the matching stage of Algorithm 1. For the GPU implementation, we always generate a maximal matching to coarsen the graph as much as possible, even if including some edges ($\{u, v\} \in E$ for which $2\,\Omega\,\omega(\{u, v\}) - \zeta(u)\,\zeta(v) < 0$) will decrease the modularity. This yields a fast algorithm, but has an adverse effect on the obtained modularity. For the CPU implementation, we only include edges $\{u, v\} \in E$ which satisfy $2\,\Omega\,\omega(\{u, v\}) - \zeta(u)\,\zeta(v) \geq 0$ in the matching, such that the modularity can only be increased by each matching stage. This yields higher modularity clusterings, but will slow down the algorithm if only a few modularity-increasing edges are available (if there are none, we perform a single matching round where we consider all edges).

Comparing Table 1 with modularities from [**17**, Table 1] for `karate` (0.412), `jazz` (0.444), `email` (0.572), and `PGPgiantcompo` (0.880), we see that Algorithm 1 generates clusterings of lesser modularity. We attribute this to the absence of a local refinement strategy in Algorithm 1, as noted in Section 4.1. The modularity of the 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.

Algorithm 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 19 million vertices and 262 million edges, the CPU generates a clustering with modularity 0.974 in 30 seconds. In particular, for clustering of nearly regular graphs (i.e. where the ratio $\big(\max_{v\in V} \deg(v)\big)/\big(\min_{v\in V} \deg(v)\big)$ is small) such as street networks, the high bandwidth of the GPU enables us to find high-quality clusterings in very little time (Table 2). Furthermore, Figure 3(a) suggests that in practice, Algorithm 1 scales linearly with the number of edges of the graph, while Figure 3(b) shows that the parallel performance of the algorithm



(a) (b)

FIGURE 3. In (a), we show the clustering time required by Algorithm 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 Algorithm 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.

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 Figure 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 that becomes 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 (i.e. where only a single of the 32 threads in a warp is still doing actual work). On the CPU, varying vertex degrees are a much smaller problem because threads are not launched in warps: they can immediately start working on a new vertex, without having to wait for other threads to finish. This results in better performance for the CPU on irregular graphs.

The most costly per-vertex operation is **compress_neighbours**, used during coarsening. We therefore expect the GPU to spend more time on coarsening than on matching for irregular graphs. For the regular graph `asia` (GPU 3.4× faster), the GPU (CPU) spends 68% (52%) of the total time on matching and 16% (41%) on coarsening. For the irregular graph `eu-2005` (CPU 4.7× faster), the GPU (CPU)

G	$\lvert V\rvert$	$\lvert E\rvert$	mod_1	t_1	$\%_1$	mod_2	t_2
karate	34	78	0.363	0.020	13	0.387	0.004
dolphins	62	159	0.453	0.027	7	0.485	0.007
chesapeake	39	170	0.186	0.024	7	0.220	0.005
lesmis	77	254	0.444	0.023	8	0.528	0.006
adjnoun	112	425	0.247	0.032	5	0.253	0.009
polbooks	105	441	0.437	0.034	6	0.472	0.008
football	115	613	0.412	0.033	5	0.455	0.009
c...metabolic	453	2,025	0.374	0.055	3	0.394	0.013
celegansneural	297	2,148	0.390	0.055	3	0.441	0.011
jazz	198	2,742	0.314	0.048	4	0.372	0.010
netscience	1,589	2,742	0.948	0.060	4	0.955	0.040
email	1,133	5,451	0.440	0.078	2	0.479	0.021
power	4,941	6,594	0.918	0.066	3	0.925	0.033
hep-th	8,361	15,751	0.795	0.093	2	0.809	0.070
polblogs	1,490	16,715	0.330	0.129	1	0.396	0.039
PGPgiantcompo	10,680	24,316	0.809	0.095	3	0.842	0.040
cond-mat	16,726	47,594	0.788	0.122	2	0.798	0.083
as-22july06	22,963	48,436	0.607	0.184	1	0.629	0.036
cond-mat-2003	31,163	120,029	0.674	0.195	2	0.690	0.103
astro-ph	16,706	121,251	0.588	0.219	1	0.611	0.085
cond-mat-2005	40,421	175,691	0.624	0.248	2	0.639	0.113
pr...Attachment	100,000	499,985	0.214	1.177	0	0.216	0.217
smallworld	100,000	499,998	0.636	0.468	2	0.663	0.175
G_n_pin_pout	100,000	501,198	0.241	0.851	1	0.246	0.231
caida...Level	192,244	609,066	0.768	0.506	2	0.791	0.198
cnr-2000	325,557	2,738,969	0.828	2.075	1	0.904	0.342
in-2004	1,382,908	13,591,473	0.946	4.403	3	0.974	1.722
eu-2005	862,664	16,138,468	0.816	8.874	1	0.890	1.854
road_central	14,081,816	16,933,413	0.996	4.562	11	0.996	13.058
road_usa	23,947,347	28,854,312	-	-.-	-	0.997	20.227
uk-2002	18,520,486	261,787,258	-	-.-	-	0.974	29.958

TABLE 1. For graphs $G = (V, E)$, this table lists the average modularities $\text{mod}_{1,2}$, (1.2), of clusterings of G generated in an average time of $t_{1,2}$ seconds by the CUDA_1 and TBB_2 implementations of Algorithm 1. The '$\%_1$' column indicates the percentage of time spent on CPU–GPU data transfer. A '-' indicates that the test system ran out of memory. Results are averaged over 16 runs. This table lists graphs from the `clustering/` category of the 10th DIMACS challenge [**1**].

spends 29% (39%) on matching and 70% (57%) on coarsening, so coarsening indeed becomes the bottleneck for the GPU when the graph is irregular.

The effectiveness of merging unmatched satellites can also be illustrated using these graphs: for `asia` the number of coarsenings performed in Algorithm 1 is reduced from 47 to 37 (1.1× speedup), while for `eu-2005` it is reduced from 10,343 to 25 (55× speedup), with similar modularities. This explains the good speedup of

G	$\|V\|$	$\|E\|$	mod_1	t_1	$\%_1$	mod_2	t_2
luxembourg	114,599	119,666	0.986	0.125	6	0.987	0.138
belgium	1,441,295	1,549,970	0.992	0.440	10	0.993	1.106
netherlands	2,216,688	2,441,238	0.994	0.615	13	0.995	1.716
italy	6,686,493	7,013,978	0.997	1.539	13	0.997	5.256
great-britain	7,733,822	8,156,517	0.997	1.793	13	0.997	5.995
germany	11,548,845	12,369,181	0.997	2.818	14	0.997	9.572
asia	11,950,757	12,711,603	0.998	2.693	15	0.998	9.325
europe	50,912,018	54,054,660	-	-.-	-	0.999	45.205
coA...Citeseer	227,320	814,134	0.837	0.420	3	0.848	0.225
coA...DBLP	299,067	977,676	0.748	0.592	3	0.761	0.279
cit...Citeseer	268,495	1,156,647	0.643	0.894	2	0.682	0.315
coP...DBLP	540,486	15,245,729	0.640	6.427	1	0.666	2.277
coP...Citeseer	434,102	16,036,720	0.746	6.490	2	0.774	2.272
kron...logn18	262,144	10,582,686	0.025	13.598	0	0.025	2.315
kron...logn19	524,288	21,780,787	0.023	28.752	0	0.023	5.007
kron...logn20	1,048,576	44,619,402	-	-.-	-	0.022	10.878
kron...logn21	2,097,152	91,040,932	-	-.-	-	0.020	23.792
333SP	3,712,815	11,108,633	0.983	2.712	7	0.984	4.117
ldoor	952,203	22,785,136	0.945	6.717	2	0.950	2.956
audikw1	943,695	38,354,076	-	-.-	-	0.857	4.878
cage15	5,154,859	47,022,346	-	-.-	-	0.682	13.758
memplus	17,758	54,196	0.635	0.160	1	0.652	0.043
rgg_n_2_20_s0	1,048,576	6,891,620	0.974	1.614	5	0.977	1.383
rgg_n_2_21_s0	2,097,152	14,487,995	0.978	3.346	4	0.980	2.760
rgg_n_2_22_s0	4,194,304	30,359,198	-	-.-	-	0.983	5.799
rgg_n_2_23_s0	8,388,608	63,501,393	-	-.-	-	0.986	12.035
rgg_n_2_24_s0	16,777,216	132,557,200	-	-.-	-	0.988	25.139

TABLE 2. Continuation of Table 1: remaining graphs of the DIMACS clustering challenge instances. From top to bottom, we list graphs from the streets/, coauthor/, kronecker/, numerical/, matrix/, walshaw/, and random/ categories.

our algorithm over [**18**] in Table 3 for eu-2005, while we do not obtain a speedup for belgium.

In the remainder of this section, we will compare our method to the existing clustering heuristic developed by Riedy et al. [**18**]. We use the same global greedy matching and coarsening scheme (Algorithm 1) to obtain clusters as [**18**]. However, our algorithm is different in the following respects. *Stopping criterion:* in [**18**] clusters are only merged if this results in an increase in modularity and if no such edges exist, the algorithm is terminated. We permit merges that decrease modularity to avoid getting stuck in a local maximum and continue coarsening as long as the modularity is within 95% of the highest encountered modularity so far. *Matching:* in [**18**] a $\frac{1}{2}$-approximation algorithm is used to generate matchings, while we use the randomised matching algorithm from [**8**]. *Coarsening:* in addition to merging matched edges, we propose a centre potential to treat star-like subgraphs efficiently, which is not done in [**18**]. *Data storage:* [**18**] uses a clever bucketing approach to

G	mod_1	t_1	mod_2	t_2	mod_O	t_O	mod_X	t_X
caida...Level	0.764	0.531	0.792	0.185	0.540	0.188	0.540	3.764
in-2004	0.955	4.554	0.976	1.887	0.475	55.986	0.488	294.420
eu-2005	0.829	9.072	0.886	1.981	0.420	90.012	0.425	1074.488
uk-2002	-	-.-	0.974	31.121	0.473	181.346	0.478	772.359
uk-2007-05	-	-.-	-	-.-	0.476	496.390	0.480	36229.531
belgium.osm	0.992	0.447	0.993	1.187	0.660	0.562	0.643	10.571
coP...DBLP	0.641	6.612	0.668	2.367	0.496	1.545	0.501	9.492
kron...logn20	0.021	59.144	0.022	13.897	0.001	538.060	0.001	8657.181
333SP	0.983	2.712	0.985	4.321	0.515	1.822	0.512	27.790
ldoor	0.944	6.799	0.950	3.071	0.542	1.348	0.611	10.510
audikw1	0.847	15.341	0.858	5.180	0.560	1.635	0.558	9.957
cage15	0.640	32.804	0.677	14.308	0.513	4.846	0.512	48.747
memplus	0.635	0.175	0.654	0.038	0.519	0.034	0.520	0.903
rgg_n_2_17_s0	0.958	0.247	0.963	0.174	0.619	0.102	0.619	1.949

TABLE 3. Comparison between Algorithm 1 and the algorithm from [**18**], using raw, single-run results for large graphs from the 10th DIMACS modularity Pareto benchmark, `http://www.cc.gatech.edu/dimacs10/results/`. Here, $\cdot_1$ and $\cdot_2$ refer to our CUDA and TBB implementations, while $\cdot_\text{O}$ and $\cdot_\text{X}$ refer to the OpenMP and Cray XMT implementations of the algorithm from [**18**]. Timings have been recorded on different test systems.

only store each edge once as a triplet, while we use adjacency lists (Section 4). A direct comparison of the performance of the DIMACS versions of both algorithms is given in Table 3. We outperform the algorithm from [**18**] in terms of quality. A fair comparison of computation times is hard because of the different test systems that have been used: we (t_1 and t_2) used two quad-core 2.4 GHz Intel Xeon E5620 processors with a Tesla C2050, while the algorithm from [**18**] used four ten-core 2.4 GHz Intel Xeon E7-8870 processors (t_O) and a Cray XMT2 (t_X).

6. Conclusion

In this paper we have presented a fine-grained shared-memory parallel algorithm for graph coarsening, Algorithm 2, suitable for both multi-core CPUs and GPUs. Through a greedy agglomerative clustering heuristic, Algorithm 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, Figure 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.

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. D. A. Bader, P. Sanders, D. Wagner, H. Meyerhenke, B. Hendrickson, D. S. Johnson, C. Walshaw, and T. G. Mattson, *10th DIMACS implementation challenge - graph partitioning and graph clustering*, 2012.
2. H. Bisgin, N. Agarwal, and X. Xu, *Does similarity breed connection? - an investigation in Blogcatalog and Last.fm communities.*, Proc of. SocialCom/PASSAT'10, 2010, pp. 570–575.
3. U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, *On modularity clustering*, IEEE Trans. Knowledge and Data Engineering **20** (2008), no. 2, 172–188.
4. T. Bui and C. Jones, *A heuristic for reducing fill-in in sparse matrix factorization*, Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing (Philadelphia, PA, USA), SIAM, 1993, pp. 445–452.
5. A. Clauset, M. E. J. Newman, and C. Moore, *Finding community structure in very large networks*, Phys. Rev. E **70** (2004), 066111.
6. T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM TOMS **38** (2011), no. 1, 1:1–1:25.
7. I. S. Duff and J. K. Reid, *Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems*, ACM TOMS **22** (1996), 227–257.
8. B. O. Fagginger Auer and R. H. Bisseling, *A GPU algorithm for greedy graph matching*, Proc. FMC II, LNCS, vol. 7174, Springer Berlin / Heidelberg, 2012, pp. 108–119.
9. B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, Proc. Supercomputing '95 (New York, NY, USA), ACM, 1995.
10. B. Hendrickson and E. Rothberg, *Improving the run time and quality of nested dissection ordering*, SIAM J. Sci. Comput. **20** (1998), no. 2, 468–489.
11. J. Hoberock and N. Bell, *Thrust: A parallel template library*, 2010, Version 1.3.0.
12. G. Karypis and V. Kumar, *Analysis of multilevel graph partitioning*, Proc. Supercomputing '95 (New York, NY, USA), ACM, 1995, p. 29.
13. B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal **49** (1970), 291–307.
14. J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, *Statistical properties of community structure in large social and information networks*, Proc. WWW '08 (New York, NY, USA), ACM, 2008, pp. 695–704.
15. M. E. J. Newman, *Fast algorithm for detecting community structure in networks*, Phys. Rev. E **69** (2004), 066133.
16. M. E. J. Newman and M. Girvan, *Finding and evaluating community structure in networks*, Phys. Rev. E **69** (2004), 026113.
17. M. Ovelgönne, A. Geyer-Schulz, and M. Stein, *Randomized greedy modularity optimization for group detection in huge social networks*, Proc. SNA-KDD '10 (Washington, DC, USA), ACM, 2010.
18. E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, *Parallel community detection for massive graphs*, Proc. PPAM11 (Torun, Poland), LNCS, vol. 7203, Springer, 2012, pp. 286–296.
19. S. E. Schaeffer, *Graph clustering*, Computer Science Review **1** (2007), no. 1, 27–64.
20. B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev. **47** (2005), no. 1, 67–95.
21. F. Zafar, M. Olano, and A. Curtis, *GPU random numbers via the tiny encryption algorithm*, Proc. HPG10 (Saarbrucken, Germany), Eurographics Association, 2010, pp. 133–141.
22. Z. Zhu, C. Wang, L. Ma, Y. Pan, and Z. Ding, *Scalable community discovery of large networks*, Proc. WAIM '08, 2008, pp. 381–388.

7. Appendix

7.1. Reformulating modularity. Our first observation is that for every cluster $C \in \mathcal{C}$, by (1.1):

$$\zeta(C) = 2\,\omega(\mathrm{int}(C)) + \omega(\mathrm{ext}(C)). \tag{7.1}$$

Now we rewrite (1.2) using the definitions we gave before:

$$\begin{aligned}
\mathrm{mod}(\mathcal{C}) &= \frac{\sum_{C\in\mathcal{C}} \omega(\mathrm{int}(C))}{\Omega} - \frac{\sum_{C\in\mathcal{C}} \zeta(C)^2}{4\,\Omega^2} \\
&= \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \left(4\,\Omega\,\omega(\mathrm{int}(C)) - \zeta(C)^2\right) \\
&\overset{(7.1)}{=} \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \left(4\,\Omega\,\left[\frac{1}{2}\,\zeta(C) - \frac{1}{2}\,\omega(\mathrm{ext}(C))\right] - \zeta(C)^2\right).
\end{aligned}$$

Therefore, we arrive at the following expression,

$$\mathrm{mod}(\mathcal{C}) = \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(\zeta(C)\,(2\,\Omega - \zeta(C)) - 2\,\Omega\,\omega(\mathrm{ext}(C))\Big). \tag{7.2}$$

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 (2.1).

7.2. 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 (2.3). Furthermore, as $\mathrm{cut}(C,C') = \mathrm{ext}(C) \cap \mathrm{ext}(C')$, we have that

$$\omega(\mathrm{ext}(C'')) = \omega(\mathrm{ext}(C)) + \omega(\mathrm{ext}(C')) - 2\,\omega(\mathrm{cut}(C,C')). \tag{7.3}$$

Using this, together with (7.2), we find that

$$\begin{aligned}
4\,\Omega^2(\mathrm{mod}(\mathcal{C}') - \mathrm{mod}(\mathcal{C})) &= -\zeta(C)\,(2\,\Omega - \zeta(C)) + 2\,\Omega\,\omega(\mathrm{ext}(C)) \\
&\quad - \zeta(C')\,(2\,\Omega - \zeta(C')) + 2\,\Omega\,\omega(\mathrm{ext}(C')) \\
&\quad + \zeta(C'')\,(2\,\Omega - \zeta(C'')) - 2\,\Omega\,\omega(\mathrm{ext}(C'')) \\
&\overset{(7.3)}{=} -\zeta(C)\,(2\,\Omega - \zeta(C)) + 2\,\Omega\,\omega(\mathrm{ext}(C)) \\
&\quad - \zeta(C')\,(2\,\Omega - \zeta(C')) + 2\,\Omega\,\omega(\mathrm{ext}(C')) \\
&\quad + (\zeta(C) + \zeta(C'))\,(2\,\Omega - (\zeta(C) + \zeta(C'))) \\
&\quad - 2\,\Omega\,\left[\omega(\mathrm{ext}(C)) + \omega(\mathrm{ext}(C')) - 2\,\omega(\mathrm{cut}(C,C'))\right] \\
&= 4\,\Omega\,\omega(\mathrm{cut}(C,C')) - 2\,\zeta(C)\,\zeta(C').
\end{aligned}$$

So merging clusters C and C' from $\mathcal{C}$ to obtain a clustering $\mathcal{C}'$, leads to a change in modularity given by (2.2).

7.3. Proof of the modularity bounds. Here, we contribute a generalisation of [**3**, Lemma 3.1] (where the bounds are established for unweighted graphs) to the weighted case. Let $G = (V, E, \omega)$ be a weighted graph and $\mathcal{C}$ a clustering of G, we will show that

$$-\frac{1}{2} \leq \operatorname{mod}(\mathcal{C}) \leq 1.$$

From (1.2),

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

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

$$\begin{aligned}
\operatorname{mod}(\mathcal{C}) &= \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(4\,\Omega\,\omega(\operatorname{int}(C)) - \zeta(C)^2\Big) \\
&\overset{(7.1)}{=} \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(4\,\Omega\,\omega(\operatorname{int}(C)) - 4\,\omega(\operatorname{int}(C))^2 - 4\,\omega(\operatorname{int}(C))\,\omega(\operatorname{ext}(C)) \\
&\qquad\qquad - \omega(\operatorname{ext}(C))^2\Big) \\
&= \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(4\,\omega(\operatorname{int}(C))\,[\Omega - \omega(\operatorname{ext}(C)) - \omega(\operatorname{int}(C))] - \omega(\operatorname{ext}(C))^2\Big) \\
&\geq \frac{1}{4\,\Omega^2} \sum_{C\in\mathcal{C}} \Big(0 - \omega(\operatorname{ext}(C))^2\Big) \quad = -\sum_{C\in\mathcal{C}} \left(\frac{\omega(\operatorname{ext}(C))}{2\,\Omega}\right)^2.
\end{aligned}$$

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

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

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 $\operatorname{mod}(\mathcal{C}) \geq -\frac{1}{2}$. The proof is completed by noting that for a single cluster, $\operatorname{mod}(\{V\}) = 0 \geq -\frac{1}{2}$.

Mathematics Institute, Utrecht University, Budapestlaan 6, 3584 CD, Utrecht, the Netherlands

E-mail address: B.O.FaggingerAuer@uu.nl

Mathematics Institute, Utrecht University, Budapestlaan 6, 3584 CD, Utrecht, the Netherlands

E-mail address: R.H.Bisseling@uu.nl