Towards Community Detection on Heterogeneous Platforms

Stijn Heldens[1(✉)], Ana Lucia Varbanescu[1], Arnau Prat-Pérez[2], and Josep-Lluis Larriba-Pey[2]

[1] Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands
stijn.heldens@student.uva.nl, a.l.varbanescu@uva.nl
[2] DAMA-UPC, Universitat Politècnica de Catalunya, Barcelona, Spain
{aprat,larri}@ac.upc.edu

Abstract. Over the last decade, community detection has become an increasingly important topic of research due to its many applications in different fields of research, such as biology and sociology. One example of a modern community detection algorithm is Scalable Community Detection (SCD), which has been shown to produce high-quality results, but its performance remains an issue on large graphs. In this work, we demonstrate how SCD can benefit from the heterogeneity offered by hybrid CPU-GPU platforms by presenting Het-SCD: a heterogeneous version of SCD which combines the larger memory capacity of the CPU with the larger computational power of the GPU. To enable this, we have designed an entirely new version of SCD which efficiently uses the fine-grained parallelism of GPUs. We report performance results on six real-world graphs (up to 1.8 B edges) and six platforms. We observe excellent performance for only the GPU (e.g., 70x speedup over sequential CPU version on graph of 117 M edges) and for combining the CPU and GPU (e.g., 40x speedup for same graph on low-end GPU with insufficient memory to store entire dataset). These results demonstrate that Het-SCD is an excellent solution for large-scale community detection, since it provides high performance while preserving the high quality of the original algorithm.

Keywords: Community detection · Heterogeneous computing · GPU computing · SCD algorithm · WCC metric

1 Introduction

In graph theory, a community is defined as a cluster of densely interconnected vertices which are sparsely connected to the rest of the network (or graph[1]). Community detection, which is the problem of partitioning a network into communities, gives us valuable insight into the structure of complex networks, such as those from biology and sociology [10]. For example, communities can correspond

[1] The terms *graph* and *network* are used interchangeably throughout this paper.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 209–220, 2015.
DOI: 10.1007/978-3-319-27308-2_18

to proteins having similar functions in protein-protein interaction networks [12], people with the same interests in social networks [15], or researchers working on the same topic in academic collaboration networks [9].

Although much research has gone into understanding the community structure of complex networks and designing community detection algorithms (see [4] for a comprehensive study), no consensus has been reached on how to detect communities and no single algorithm has become universally accepted. Additionally, little effort has been placed on performance, which becomes a problem when the size of these networks grows. Analyzing massive networks consisting of millions, or even billions, of edges can take minutes, or even hours, using state-of-the-art community detection algorithms. Reducing the processing time of these algorithms remains desirable.

Scalable Community Detection (SCD, Sect. 2.2) is an example of a modern algorithm community which has been shown to produce high-quality results. SCD partitions a network into communities by greedily maximizing the *Weighted Community Clustering* (WCC, Sect. 2.1) metric, a novel community quality metric based on triangle counting. SCD has been designed to be highly parallel to run efficiently on modern parallel platforms.

In this work, we show the potential of this algorithm for massively parallel architectures by presenting the first version of SCD specifically designed for GPUs. This version runs the most computationally expensive phase of the algorithm entirely on the GPU. This solution leads to high performance, but requires the entire network to fit into the memory of the device. To tackle this limitation, we have extended this version to *Het-SCD*: a heterogeneous version of SCD which runs on hybrid CPU-GPU platforms. Het-SCD attempts to process as many vertices as possible on one or more GPUs and processes the remaining vertices on the CPU. By doing this, we effectively combine the larger memory capacity of the CPU with the larger computational power of the GPU.

We have evaluated our work on six real-world datasets from the SNAP repository [7], the largest having 1.8 billion edges. The results show that only using the GPU allows Het-SCD to obtain orders of magnitude speedup compared to a sequential CPU implementation. Additionally, our results show that using the GPU is still beneficial for performance even if the size of the network exceeds GPU memory and a fraction of the vertices needs to be processed on the CPU.

2 Background

The *Scalable Community Detection* (SCD) algorithm partitions an undirected unweighted network into communities by maximizing the *Weighted Community Clustering* (WCC) metric. In this section, we briefly introduce WCC and SCD. For more detailed descriptions, we refer to [13,14], respectively.

Fig. 1. The three steps of the SCD algorithm (Sect. 2.2). Het-SCD focuses on the partition refinement (Sect. 3.1).

2.1 The WCC Metric

WCC is a metric that measures the quality of a partitioning of a network into communities. It is based on the intuition that vertices close more triangles with vertices inside the same community than with those outside the community.

Given an undirected unweighted graph $G = (V, E)$, a vertex $x \in V$, and a community $C \subseteq V$, let $t(x, C)$ be the number of triangles that vertex x closes with vertices in C and let $vt(x, C)$ be the number of vertices in C which close at least one triangle with x and a third vertex from C. The cohesion of vertex x to community C is defined as in Eq. 1.

$$WCC(x, C) = \begin{cases} \frac{t(x,C)}{t(x,V)} \cdot \frac{vt(x,V)}{|C \setminus \{x\}| + vt(x,V) - vt(x,C)} & \text{if } t(x, V) \neq 0 \\ 0 & \text{if } t(x, V) = 0 \end{cases} \quad (1)$$

Now, let $\mathcal{P} = \{C_1, \ldots, C_k\}$ be a partition of V, i.e., $C_1 \cup \ldots \cup C_k = V$ and $C_i \cap C_j = \emptyset$ if $i \neq j$. Define C^x as the community to which vertex x belongs. The WCC of $\mathcal{P}$ is defined as in the average $WCC(x, C^x)$ over all vertices x.

2.2 The Scalable Community Detection Algorithm

The SCD algorithm takes a graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, and partitions G into communities by greedily maximizing WCC. Figure 1 shows the three steps of the algorithm: preprocessing, initial partition and partition refinement.

Preprocessing. During preprocessing, the number of triangles closed by each edge is counted. Using this data, the number of triangles closed by each vertex calculated. Edges which do not close any triangles are deleted from the graph since they do not affect the WCC. Vertices which become isolated after this step are also removed: they will be assigned to new singleton communities afterwards. The time complexity of this stage is $\mathcal{O}(m \log n)$ [14], assuming a quasi-linear relation between n and m, i.e., $\mathcal{O}(m) \sim \mathcal{O}(n \log n)$.

Initial Parition. A fast heuristic [14] is used to assign the vertices to initial communities. The vertices are visited in descending order of clustering coefficient[2] (calculated using data from preprocessing) and assigned to newly created

[2] The clustering coefficient cc of a vertex x is defined as $cc = 2t/d(d-1)$ where t is the number of triangles x closes and d is the degree of x.

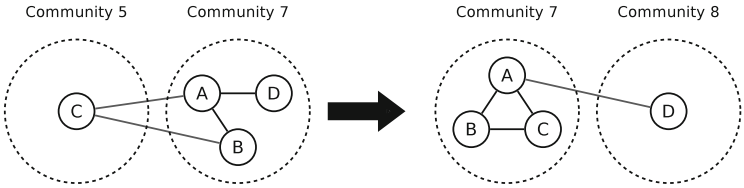

Fig. 2. Results of one refinement iteration: vertices A and B stay in community 7 (action (1)), vertex C is transferred to community 7 (action (2)), and vertex D is placed in a new community (action (3)).

communities. The most expensive operation of this phase is sorting the vertices, which requires $\mathcal{O}(n \log n)$ time.

Partition Refinement. The initial partition serves as input to the refinement phase in which the WCC is iteratively improved. Figure 2 demonstrates one iteration of this phase. In each iteration, all vertices performs the action which leads to the largest increase in WCC. There are three possible types of actions:

(1) **No action**: the vertex stays in its current community.
(2) **Remove**: the vertex is removed from its current community and placed alone in a newly created community.
(3) **Transfer:** the vertex is transferred from its current community to the community of one of its neighbors.

Action (1) does not affect the WCC. Actions (2) and (3) do affect the WCC, but accurately computing the improvement is computationally expensive since it requires recounting many internal triangles in the network. Prat et al. [14] proposed a constant-time approximation model to estimate the impact of these two actions on the WCC. This model allows one to estimate the improvement in WCC when adding/removing a vertex x to/from community C, given the community statistics of C (number of vertices and number of boundary/internal edges), the number of edges from vertex x towards community C and the average clustering coefficient of the entire network.

Note that all vertices select and apply their best action *in parallel*, thus exposing massive parallelism. At the end of each iteration, the WCC of the resulting partition is calculated and the algorithm continues with a new iteration unless the overall increase in WCC is less than a given threshold. Each iteration takes $\mathcal{O}(m \log n)$ time [14]. The number of iterations required to reach convergence depends on the size and the topology of the graph.

3 Design and Implementation

We have designed and implemented Het-SCD: a heterogeneous version of SCD for CPU-GPU platforms. First, we discuss the massively parallel version of SCD which we designed specifically for GPUs. Next, we discuss how we extended this version to support hybrid CPU-GPU platforms. Finally, we discuss how to automatically distribute the vertices of a network over multiple devices.

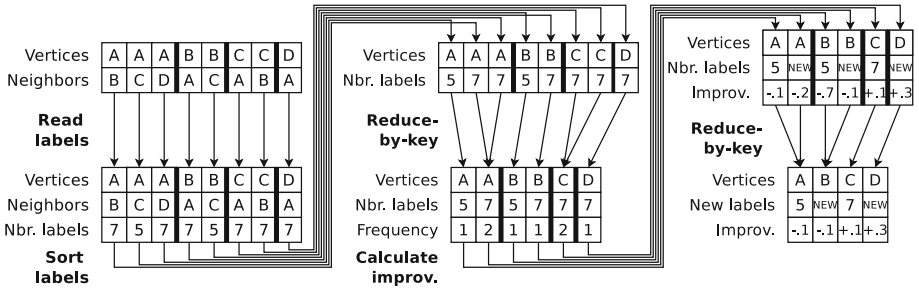

Fig. 3. Procedure used to determine new labels for vertices of graph in Fig. 2.

3.1 The Massively Parallel Version

For our current massively parallel version of SCD, we focused our attention to the partition refinement phase (see Fig. 1) since this is the most expensive phase of the algorithm. After the preprocessing and initial partition phases (performed on the CPU), each vertex is assigned a label corresponding to the identifier of its community. This labeling is transferred to the GPU, together with the graph in *compressed sparse row* format, and the data collected during preprocessing.

Each refinement iteration is performed entirely on the GPU. The three steps of each iteration are: update the labels, collect community statistics, and calculate the WCC.

Update Labels. Updating the vertices of the labels is done by, for each vertex, evaluating all possible actions (Sect. 2.2) and applying the action which leads to the largest increase in WCC. A straight-forward solution to do this using a *vertex-centric* approach, i.e., assigning each vertex to a thread. Each vertex is updated by iterating over its neighbors, keeping track of the frequencies its neighboring labels using an associate array, evaluating the benefit of each possible action, and applying the most beneficial action.

However, a vertex-centric approach is not suitable for massively parallel architectures for a number of reasons. First, it leads to severe **load imbalance**: The work per vertex is determined by its degree and the degree distribution of real-world networks usually follows a *power law* [9], i.e., many low-degree vertices and few high-degree vertices. Second, the amount of **parallelism** is limited: evaluating all possible actions for a single vertex is done sequentially even though it can be parallelized. Third, associative arrays, such as binary trees or hash tables, are often not efficient on massively parallel architectures since they require **dynamic memory allocation** and have poor **memory coalescing**.

To tackle these challenges, we have designed, from the ground up, a new produce to update the vertices using an *action-centric* approach, i.e., all actions for all vertices are evaluated in parallel. Our approach is based on generic global parallel primitives, such as `sort` and `reduce`, to obtain high hardware utilization since many of these primitives have been extensively researched.

Our procedure starts with a sorted directed edge list of the network. For each edge, the label of the incoming endpoint is read, resulting in a list of vertex-label pairs. These pairs are sorted using a global *segmented sort*[3] to place matching pairs adjacent to each other. The frequency of each vertex-label pair (v, C) corresponds to the number of edges between vertex v and community C. A reduce-by-key[4] operation is used to count the frequency of each pair.

For all vertex-label-frequency triples (v, C, f), the improvement in WCC that results from transferring vertex v to community C (action (2)) is calculated using the approximation model (Sect. 2.2). If vertex v is already assigned to community C, the improvement when removing v from C (action (3)) is calculated instead. Finally, another reduce-by-key operation is used select the best action for each vertex. Every vertex adopts the resulting label if the corresponding improvement is positive, other it keeps its current label (action (1)).

For our prototype, we used the reduce-by-key and segemented sort primitives from the Modern GPU library [1].

Collect Community Statistics. Changing the labels of the vertices affects the community statistics (number of vertices and number of internal/boundary edges), so these need to be recalculated. The statistics are updated by processing all vertices and edges in parallel and incrementing counters using atomics.

Calculate WCC. The final step of each iteration is calculating the new WCC to determine whether another iteration is necessary. The only values from Eq. 1 which are not known for each vertex x are $t(x, C^x)$ and $vt(x, C^x)$. The problem of computing these values is similar to the well-studied problem of triangle counting, with the exception that we are only interested in *internal* triangles, i.e., triangles for which all endpoints lie within the same community.

A possible solution to this problem is to intersect the adjacency list of the endpoints of every edge and check, for each triangle found, whether it is internal. However, this method is inefficient since, in practice, only a small fraction of all triangles is internal. Therefore, we prune the graph by first removing all inter-community edges, thus making all triangles internal, and intersect the pruned adjacency lists. This approach exposes a lot of parallelism since both pruning the graph and intersecting adjacency lists can be done in parallel.

Once the WCC of every vertex has been calculated, the average is calculated using a reduction operation and the result is transferred back to the host, which decides whether another iteration is necessary.

3.2 The Heterogeneous Version

The massively parallel version of SCD presented above is able to perform the entire refinement phase on the GPU, but it required the entire network to be

[3] Segmented sort takes a list of consecutive non-uniform segments and sorts each one.
[4] Reduce-by-key divides a list of key-value pairs into segments with matching consecutive keys and reduces the values in each segment to a single value.

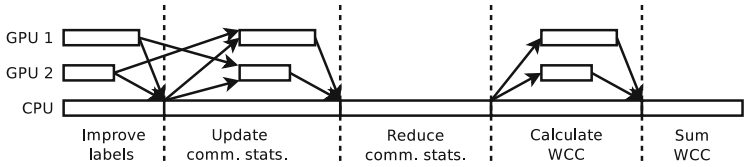

Fig. 4. The steps of refinement phase for our heterogeneous version when using two GPUs.

stored in GPU memory thus limiting the size of the networks that can be processed. To tackle this challenge, we extended this version to *Het-SCD*: a heterogeneous version of SCD which processes as many vertices as possible on one or more GPUs and the remaining vertices on the CPU.

Het-SCD requires each vertex to be assigned to either the CPU or a GPU, resulting in a partitioning of the network into components. Vertices that belong to a component are known as the *core* vertices of the component and adjacent vertices are known as *halo* vertices. Each vertex is thus a core vertex in exactly one component and can duplicated as a halo vertex in several components.

At the start of the refinement phase, subgraphs consisting of core and halo vertices are transferred to the GPUs. Processing the vertices on CPU and GPU in parallel is possible since all steps of the refinement phase are vertex-centric and the vertices of the network are distributed over the available devices. Our GPU implementation is written in CUDA [11] and our CPU implementation was provided by Prat et al. [14] and has been parallelized using OpenMP. The vertices which are processed on GPUs are masked out in the OpenMP implementation.

The CPU and GPUs need to communicate after each step of every refinement iteration as illustrated in Fig. 4. After updating the labels, all devices need to exchange data to update the labels of their halo vertices. After collecting the statistics of the communities in its component, each GPU transfers their statistics to the CPU, the CPU sums the results and transfers the final statistics back to the GPUs. After calculating the average WCC of its vertices, each GPU sends its result to the CPU which computes the overall average WCC.

3.3 Automatic Partitioning

Het-SCD requires, as input, a partitioning of the input network over the available devices without exceeding the memory of any device. This partitioning should have a low cut to minimize the number of halo vertices and thus the storage and communication cost of these vertices. Additionally, the edge density and triangle distribution of all components should be balanced to evenly distribute the work.

We used METIS [6] to automatically partition the network. Each vertex is assigned its degree and number of triangles it closes as weights to evenly distribute the edges and triangles. Experiments show that METIS yields balanced partitionings, but its performance is low. For example, when evenly splitting

Table 1. The platforms used for our experiments.

Platform	GPU name	CUDA	SM count	Clock (Mhz)	Mem (MB)	Bandwidth (GB/s)	Host CPU (Intel Xeon)
A	C2050 (Fermi)	2.0	14	575	2688	144.0	E5620
B	GTX480 (Fermi)	2.0	15	700	1536	177.4	E5620
C	GTX580 (Fermi)	2.0	16	772	3072	193.0	X5650
D	GTX680 (Kepler)	3.0	8	1006	2048	192.3	E5620
E	Tesla K20m (Kepler)	3.5	13	706	5120	208.0	E5-2620
F	GTX-Titan (Kepler)	3.5	14	837	6144	288.0	E5620

Table 2. The networks used for our experiments. **Size** is the size of the graph while **Footprint** includes both the graph and auxiliary data structures in GPU memory.

Name	Type	Vertices (mil.)	Edges (mil.)	Size (MB)	Footprint (MB)
Amazon	Co-purchasing network	0.335	0.926	23.0	54.9
DBLP	Coauthorships network	0.317	1.050	27.2	69.8
YouTube	Social network	1.135	2.988	30.9	92.9
LiveJournal	Social network	3.998	34.681	562.8	1778.6
Orkut	Social network	3.072	117.185	1720.0	5026.6
Friendster	Social network	65.608	1806.067	27812.4	-

LiveJournal (Table 2) into two components, the differences in the number of vertices, edges and triangles are all less than 1 %, but the run-time is 35 s.

4 Evaluation

We have evaluated our solution using six platforms (Table 1) and six networks (Table 2). The networks have been chosen from SNAP [7] since they contain known ground-truth communities, which makes these datasets suitable candidates for a community detection algorithm. The source code and a full report of our results are available online[5]. In this section, we summarize the most important performance results. All our results are averaged over 5 runs. Error bars have been omitted since results were found to be stable. The threshold for WCC improvement was set to 1 % [14].

4.1 The GPU Version

Figure 5 shows the average speedup per iteration of the refinement process for the different GPUs over a serial version of SCD which is based on previous work [14]. This version has also been parallelized for multi-core processors using OpenMP (OMP), and the results for sixteen threads on an Intel E5620 dual-quad-core CPU have been included for comparison.

[5] http://github.com/Het-SCD.

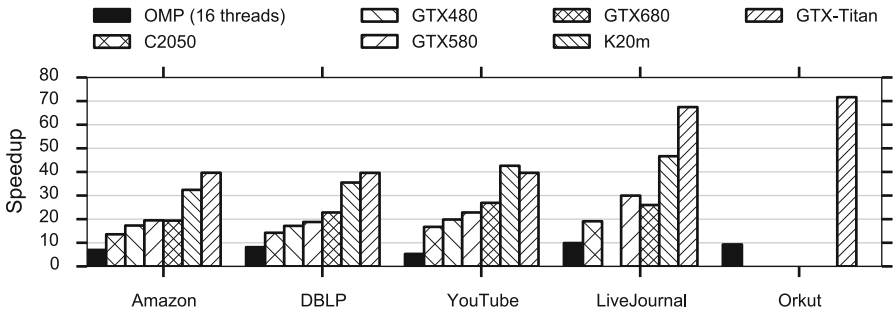

Fig. 5. Average speedup per iteration of the refinement process over serial version on Intel E5620. Missing bars indicate failures due to insufficient memory on the GPU.

The results clearly show that the GPU version is always significantly faster than both the serial and multi-threaded version, regardless of the network or the GPU used. Even the least powerful GPU, the C2050, obtains speedups between 10x and 20x on every graph. The most powerful GPU, the GTX-Titan, obtains speedups from 40x on Amazon up to a massive 70x on Orkut. This is the result of the massive parallelism and increased bandwidth offered by these GPUs.

However, the results also show that the amount of GPU memory is a realistic limitation which cannot be ignored. Friendster has not been included since it does not fit into the memory of any GPU. Orkut is also too large for most of our GPUs and can only be processed by the GTX-Titan. LiveJournal can be processed by most of the GPUs, except the GTX480.

4.2 The Heterogenous Version

The heterogeneous version removes the memory restriction by processing as many vertices as possible on GPUs and the remaining vertices on the CPU.

Figure 6 shows the average time per iteration of the refinement phase for LiveJournal and Orkut when combing CPU and GPU. The results show that Het-SCD can be used to process networks which cannot be processed only by the GPU since they exceed GPU memory. For example, LiveJournal can be processed on the GTX480 by assigning only 20 % of the vertices to the CPU, resulting in a speedup of 3.1x compared to the multi-threaded version and 30.7x compared to the serial version. Orkut can be processed on the K20m by also assigning only 20 % of the vertices to the host, giving a speedup of 4.5x and 41.5x compared to the multi-threaded and serial version, respectively.

The heterogeneous version can also use multiple GPUs simultaneously. Figure 7 shows the speedup for LiveJournal and Orkut when using multiple GTX580 GPUs. For LiveJournal, the speedup is significant up to 4 GPUs. Using more GPUs decreases the speedup due to communication overhead. For Orkut, we have predicted the single GPU performance using information from Fig. 6 since it exceeds the memory of a single GTX580. Using 6 GPUs provides enough memory to process the graph and more GPUs give better performance.

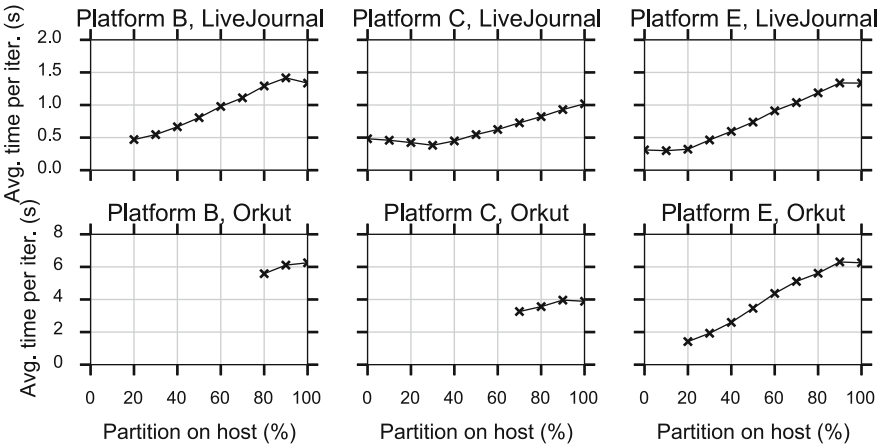

Fig. 6. Average time per iteration of the heterogeneous version for two networks (LiveJournal and Orkut) and three platforms. The number of threads used on the CPU was set to the number of available virtual cores. Missing points indicate failures due to insufficient memory on the GPU.

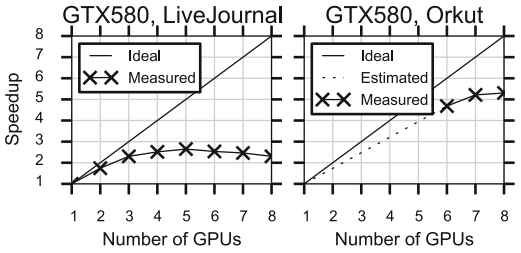

Fig. 7. The speedup when dividing the vertices over multiple GPUs.

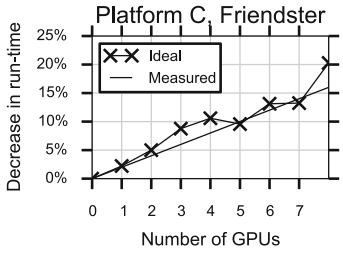

Fig. 8. Speedup when combining CPU and GPUs for Friendster.

We also experimented if Friendster, having 1.8 billion edges, can be processed using our solution. Using a back-of-the-envelope calculation, we estimated that 2 % of Friendster can be placed on a single GTX580. We processed this graph by assigning 2 % of the vertices to each GPU and the remaining vertices to the CPU. Figure 8 shows that every added GPU indeed decreases the run-time by roughly additionally 2 %. When using 8 GPUs, the overall run-time is reduced by 20 % (6.1 s per iteration).

4.3 End-to-End Performance

Up to this point, we have only reported the performance of the refinement phase, which is the part we have accelerated. However, the complete Het-SCD application also reads the network from disk, performs preprocessing, creates the initial

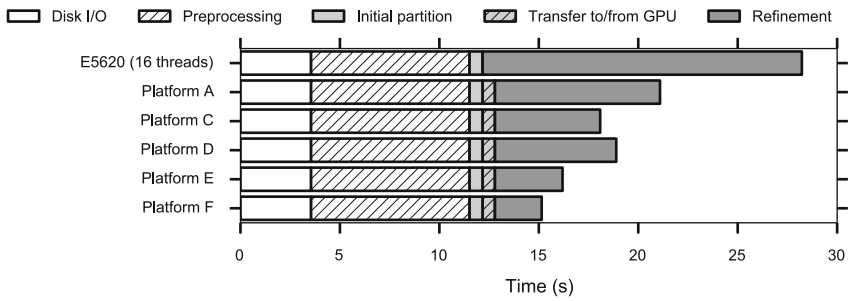

Fig. 9. The end-to-end execution time for LiveJournal.

communities and transfers data between main memory and GPU memory. These steps have not been accelerated by Het-SCD. An example of a complete execution profile for LiveJournal is shown in Fig. 9, the other datasets show a similar result. The figure shows that the refinement phase accounts for more than 55 % of the total execution time when using 16 threads. When using the GPU, this time is reduced by a factor of between 2x and 7x (depending on the chosen GPU), reducing the total execution time by between 25 % and 50 %, which is significant. We propose that parallelizing the preprocessing step and reducing disk I/O should complement Het-SCD, but consider this effort to be out of the scope of this paper.

5 Related Work

Many community detection algorithms have been proposed [4], but only few have been implemented on GPUs. Soman et al. [16] proposed a GPU implementation of an algorithm based on label propagation. Cheong et al. [3] and Staudt et al [17] designed a GPU implementation around the Louvain method [2], an algorithm based on modularity maximization. However, label propagation suffers from "label epidemic" where some labels manage to "plague" the network [8] and modularity maximization suffers from the resolution limit [5]. SCD is a novel algorithm which has been proven to be robust and find meaningful and cohesive communities [14].

Staudt et al [17] also extended their Louvain-based algorithm to support hybrid CPU-GPUs platforms. However, they achieved this by partitioning the graph, running the algorithm on each subgraph, and combining the results. This modification lowers the quality of the original algorithm. We designed Het-SCD to preserve the original algorithm to ensure its high quality, while providing higher performance.

6 Conclusion and Future Work

High-quality community detection in large networks is becoming an important requirement for modern data mining applications. In this work, we target both

the performance and scale of the problem: we design the first massively parallel version of SCD, a high-quality community detection algorithm, and an extension of this version to a flexible, large-scale heterogeneous version. Our experimental results demonstrate (1) the superior performance of the GPU version compared to both the sequential and the parallel CPU execution, and (2) the ability of the heterogeneous version to achieve significant performance gains by processing large graphs on the CPU and multiple GPUs in parallel.

There are further steps to be taken to improve Het-SCD. So far, we have only accelerated the refinement phase on the GPU. In the near future, we plan to implement a parallel version of the preprocessing phase and initial partition as well. Finally, the impact of the partitioning on the performance of Het-SCD must be explored. While METIS yields balanced partitions with a low cut size, its performance is low. Designing a custom partition algorithm specifically for Het-SCD might give higher performance.

References

1. Baxter, S.: Modern GPU. http://moderngpu.com/ (2013)
2. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. J. Stat. Mech: Theory Exp. **2008**(10), P10008 (2008)
3. Cheong, C.Y., Huynh, H.P., Lo, D., Goh, R.S.M.: Hierarchical parallel algorithm for modularity-based community detection using GPUs. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 775–787. Springer, Heidelberg (2013)
4. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3), 75–174 (2010)
5. Fortunato, S., Barthélemy, M.: Resolution limit in community detection. Proc. Nat. Acad. Sci. **104**(1), 36–41 (2007)
6. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
7. Leskovec, J., Krevl, A.: SNAP datasets: stanford large network dataset collection (2014). http://snap.stanford.edu/data
8. Leung, I.X., Hui, P., Lio, P., Crowcroft, J.: Towards real-time community detection in large networks. Phys. Rev. E **79**(6), 066107 (2009)
9. Newman, M.E.: The structure and function of complex networks. SIAM Rev. **45**(2), 167–256 (2003)
10. Newman, M.E.: Detecting community structure in networks. Eur. Phys. J. B Condens. Matter Complex Syst. **38**(2), 321–330 (2004)
11. NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2011)
12. Palla, G., Derényi, I., Farkas, I., Vicsek, T.: Uncovering the overlapping community structure of complex networks in nature and society. Nature **435**, 814–818 (2005)
13. Prat-Pérez, A., Dominguez-Sal, D., Brunat, J.M., Larriba-Pey, J.L.: Shaping communities out of triangles. In: CIKM 2012. ACM (2012)
14. Prat-Pérez, A., Dominguez-Sal, D., Larriba-Pey, J.L.: High quality, scalable and parallel community detection for large real graphs. In: WWW 2014 (2014)
15. Scott, J.: Social Network Analysis. Sage, London (2012)
16. Soman, J., Narang, A.: Fast community detection algorithm with GPUs and multicore architectures. In: IPDPS 2011. IEEE (2011)
17. Staudt, C., Meyerhenke, H.: Engineering parallel algorithms for community detection in massive networks. IEEE TPDS **PP**(99), 1 (2015)