Hierarchical Parallel Algorithm for Modularity-Based Community Detection Using GPUs

Chun Yew Cheong[1], Huynh Phung Huynh[1], David Lo[2], and Rick Siow Mong Goh[1]

[1] Institute of High Performance Computing, A*STAR, Singapore
{cheongcy,huynhph,gohsm}@ihpc.a-star.edu.sg
[2] Singapore Management University, Singapore
davidlo@smu.edu.sg

Abstract. This paper describes the design of a hierarchical parallel algorithm for accelerating community detection which involves partitioning a network into communities of densely connected nodes. The algorithm is based on the Louvain method developed at the Université Catholique de Louvain, which uses modularity to measure community quality and has been successfully applied on many different types of networks. The proposed hierarchical parallel algorithm targets three levels of parallelism in the Louvain method and it has been implemented on single-GPU and multi-GPU architectures. Benchmarking results on several large web-based networks and popular social networks show that on top of offering speedups of up to 5x, the single-GPU version is able to find better quality communities. On average, the multi-GPU version provides an additional 2x speedup over the single-GPU version but with a 3% degradation in community quality.

Keywords: Community detection, parallel algorithm, GPU, social networks.

1 Introduction

Detecting community structure has attracted increasing attention [1] with the recent rise of social networks such as Facebook and Twitter. Community detection involves clustering highly connected nodes in a network into communities. For social networks, community detection can be applied to online marketing campaigns such as recommendation systems and viral marketing strategies. Besides social networks, community detection has many applications in other areas such as finding webpages that have the same topic in WWW [2], identifying communities for contact tracing in the event of infectious disease outbreak [3], and identifying a group of friends in a mobile network [4].

Nowadays, networks with hundreds of millions of nodes and links are common and their sizes continue to increase. Community detection on these huge networks will take a large amount of time. This will limit the quality of community information extracted due to significant computational complexity. In order to cope with larger networks as well as perform more complex analysis with faster response time, there is a need to accelerate the performance of the core kernel of the community detection algorithm.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 775–787, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Graphics Processing Unit (GPU) acceleration has been the current trend in the high performance computing community. Each GPU can have up to 2000 physical processing cores running in parallel [5]. On top of these processing cores, tens of thousands of software threads are concurrently executed in interleaved fashion to maximize the performance. With its massively parallel computing power, GPU is potentially a suitable candidate for accelerating community detection algorithms for large networks. However, mapping a community detection algorithm to GPU faces some challenging problems. Firstly, processing a large graph requires a large amount of data communication among different functions of the algorithm but data communications between CPU and GPU or within GPU memory hierarchy are very expensive. Secondly, core computation of the community detection algorithm is quite diversified among nodes and communities but GPU execution uses a Single Instruction Multiple Threads (SIMT) model. In the SIMT model, all threads execute the same instructions at a time step. Finally, there is a need to efficiently utilize the heterogeneous computing power in GPU, multi-GPU, and multi-core systems. To address the above challenges, an efficient implementation of the community detection algorithm using GPU as accelerator is proposed in this paper. The proposed parallel community detection algorithm is based on the highly-cited Louvain method [4]. The key contributions of this paper are summarized:

1. Proposing the first parallel community detection algorithm based on the Louvain method.
2. Accelerating the parallel Louvain version on single GPU platform.
3. Further accelerating the parallel Louvain version on a multi-core with multi-GPU architecture.
4. Comprehensive experiments on different networks: social networks (Facebook [6], Twitter (extracted by Twitter API), Orkut [7], LiveJournal [7], Flickr [8]) and web-based networks (uk-2005 [9], webbase-2001 [10]).

2 Background

2.1 Modularity-Based Community Detection

The community detection problem has been well-studied in the literature. Existing algorithms can be classified into a few major classes: divisive algorithms [11], modularity-based methods [12], dynamic methods [13], and spectral methods [14]. A more complete summary about community detection algorithms can be found in the survey paper [1]. In this paper, the focus is on modularity-based methods which use modularity as a measure of the quality of a community. Modularity, proposed by Newman [12], is the comparison between the actual density of links in a sub-graph (community) and the density one is expected to have in the sub-graph if the vertices of the sub-graph were attached arbitrarily. For a directed graph, the modularity Q is defined as follows:

$$Q = \frac{1}{m} \sum_{i,j \in V} \left(A_{ij} - \frac{k_i^{out} k_j^{in}}{m} \right) \delta(c_i, c_j) \quad (1)$$

where V is the set of nodes in the network, A_{ij} is the weight of the link between nodes i and j, k_i^{out} and k_j^{in} are the sum of the weights of the outgoing and incoming links of i and j, respectively, c_i is the community that node i is in, the δ function $\delta(u,v)$ is 1 if u is equal to v and 0 otherwise, and m is the sum of the weights of all the links in the network. From (1), the modularity of a network ranges between 0 and 1, with a higher value indicating a stronger community structure.

Among modularity-based methods, the Louvain method [4] is well-known to be able to perform fast community detection. It has been successfully applied on many different types of networks [15-17]. Instead of computing modularity for the whole network as in (1), which is computationally expensive, the Louvain method introduced the gain in modularity of moving a node i into a community C, which is computed as follows:

$$\Delta Q = \frac{1}{m}\left(k_{i,C} + k_{C,i} - \frac{k_i^{out}k_C^{in} + k_i^{in}k_C^{out}}{m}\right) \quad (2)$$

where $k_{i,C}$ is the sum of the weights of the links from i to the nodes in C, $k_{C,i}$ is the sum of the weights of the links from the nodes in C to i, and k_C^{in} and k_C^{out} are the sums of the weights of the incoming and outgoing links of all the nodes in C, respectively. k_i^{out}, k_i^{in}, and m are as defined in (1).

2.2 GPU Computing

A GPU supports massive parallelism through a number of streaming multi-processors (SMs), each consisting of a number of physical processing cores running in SIMT mode. Parallel software threads are grouped into thread blocks which run on each of the available SMs. There are typically more software threads than there are physical processing cores. In order to efficiently schedule the large number of software threads on SMs, 32 threads are statically grouped into scheduling units, referred to as warps in the NVIDIA literature. Warps execute in lockstep, and if one or more threads in a warp wait for data to be ready, the entire warp has to wait as well. A hardware scheduler will then select another ready warp for execution. Recently, Soman et al. [18] proposed a community detection algorithm based on label propagation and mapped it to GPU platform using some standard GPU primitives such as Bitonic sort.

3 The Louvain Method for Community Detection

3.1 Description of the Louvain Method

The Louvain method [4] is a greedy optimization method for community detection. It partitions a network into communities by maximizing the modularity of the partition. In essence, the Louvain method consists of two phases, modularity optimization and community aggregation. In the modularity optimization phase, each node of the network is initially assigned to its own community, i.e. the number of communities is equal to the number of nodes. Each node is then considered, in turn, if it would stay in its original community or move to one of its neighboring communities. This is done

by removing the node from its original community, computing the gain in modularity if the node were inserted into each of its neighboring communities, and moving the node to the community with the maximum positive gain. Each cycle of this process through all the nodes in the network is referred to as an iteration. The modularity optimization phase terminates when no improvement in modularity can be achieved. At the end of the modularity optimization phase, the network is partitioned into a number of communities. Next, the community aggregation phase involves building a new, but smaller, network by aggregating nodes in the original network that belong to the same community such that the nodes in the new network are the communities at the end of the preceding modularity optimization phase. The weight of the link between two nodes in this new network is the total weight of the links between the nodes of the two corresponding communities in the original network. The links between the nodes of the same community become self-loops of the corresponding node in the new network. With the new network, the modularity optimization phase can then be applied again and the two phases iterate until no improvement in modularity can be achieved. Each application of the modularity optimization phase followed by the community aggregation phase is referred to as a pass.

3.2 Profiling of the Louvain Method

In order to identify the computational bottlenecks that should be targeted when parallelizing the Louvain method to speed up its computation, a profiling of the method was conducted on a web-based network. The network is a sub-network of the .uk domain and it has 16 million nodes and 287 million links. The profiling results are shown in Fig. 1. In order to highlight the contribution of this paper, the profiling results, as well as all subsequent timing results, will only consider the computation time spent on community detection, while I/O times are omitted.

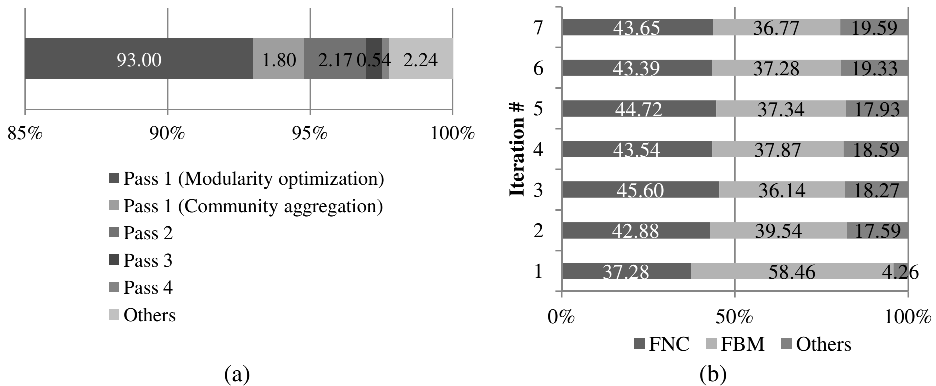


Fig. 1. Profiling results: (a) Percentage of computation time spent in each pass and (b) breakdown of computation time of each iteration of the modularity optimization phase of the first pass

Figure 1(a) shows that 94.8% of the computation time is spent in the first pass of the Louvain method. This result is expected since all the nodes in the network are

considered in the first pass, while the other passes process much smaller networks due to the community aggregation phase. Figure 1(a) also shows that 93% of the computation time is spent in the modularity optimization phase of the first pass. It can be seen in Fig. 1(b) that within each of the seven iterations of the modularity optimization phase of the first pass, approximately 80% of the computation time is spent on two main components of the modularity optimization phase. The first component, referred to as Find Neighboring Communities (FNC), involves computing the set of unique neighboring communities for each node i. For each neighboring community C, $k_{i,C}$ and $k_{C,i}$, which are needed for the calculation of the gain in modularity in (2), are also computed at the same time. The other component, referred to as Find Best Move (FBM), computes the gain in modularity of moving each node into each of its neighboring communities and then moving the node to the community that gives the maximum positive gain.

From the profiling results, it is clear that the modularity optimization phase of the first pass of the Louvain method is the main computational bottleneck that should be targeted when parallelizing the algorithm. In order to speed up the modularity optimization phase of the first pass, it is critical to accelerate the computation of FNC and FBM.

4 Hierarchical Parallel Algorithm

The hierarchical parallel algorithm for community detection proposed in this paper targets three levels of parallelism in the Louvain method to speed up its computation.

At the highest level, the original network is partitioned into a number of sub-networks and a set of removed links which consists of the links that join nodes residing in different sub-networks. The Louvain method can then be applied to solve the community detection problem in each of the sub-networks in parallel. After this, the resulting networks are combined into a single network using the removed links, and then the Louvain method is applied once more on this combined network to obtain the final community results. On top of decomposing the original network into sub-networks and processing them in parallel, the effectiveness of this level of parallelism also stems from the fact that the combined network, obtained from the resulting networks after processing the sub-networks in parallel, is typically a few orders of magnitude smaller than the original network due to the community aggregation phase of the Louvain method.

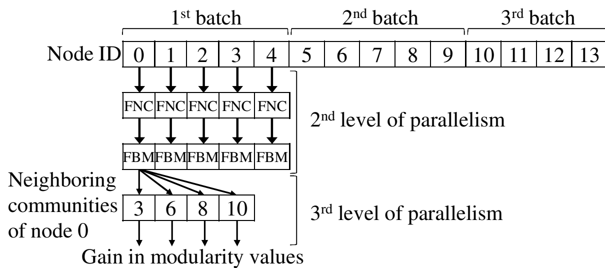


Fig. 2. Illustration of the second and third levels of parallelism

The second level of parallelism involves visiting nodes in parallel during each iteration of the modularity optimization phase. In the original Louvain method, nodes are visited sequentially in each iteration, where each node visit involves applying FNC to obtain the set of neighboring communities of the node, followed by FBM to move the node to the community that results in the maximum positive gain in modularity. As shown in Fig. 2, this level of parallelism suggests the division of nodes in the network into batches, with the nodes in each batch being processed in parallel. It is to be noted that the computations for visiting two nodes in a batch may not be independent since one of the nodes may be in the neighboring community of the other node. While it is possible to find mutually independent batches, it would incur additional computational cost. In this paper, this inaccuracy is allowed but only atomic updates to a node's community status are permitted.

The third and lowest level of parallelism involves computing the gain in modularity of inserting a node into each of its neighboring communities in parallel (See Fig. 2). This level of parallelism is intuitive and would be effective when a node has a large number of neighboring communities.

5 Mapping to GPU

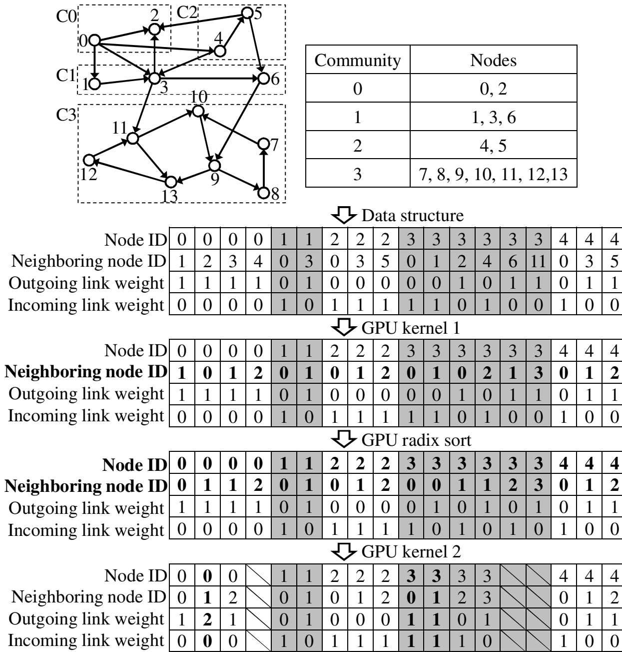
This section describes how the three levels of parallelism proposed in the previous section for the Louvain method are implemented on the GPU.

5.1 Mapping of Find Neighboring Communities to GPU

As described in Section 3.2, Find Neighboring Communities (FNC) performs two functions. It not only finds the set of neighboring communities for each node i but for each neighboring community C of i, it also computes $k_{i,C}$ and $k_{C,i}$ which are needed for the calculation of the gain in modularity as given in (2).

An example to illustrate the mapping of FNC to GPU is shown in Fig. 3. In the figure, the current community status of the network is shown. It can also be seen in the figure that the network is represented by an array of structures. Each structure consists of four elements – the node ID, the neighboring node ID, the outgoing link weight, and the incoming link weight. For example, the first column in the data structure indicates that node 0 has an outgoing link of weight 1 to its neighboring node 1. Only five nodes, i.e. node 0 to node 4, are considered in the data structure in the figure as it is assumed that nodes are processed in batches of five in the second level of parallelism as shown in Fig. 2. The data for the five nodes are combined into a single array and copied to the host memory of the GPU to reduce communication overhead for each memory copy instruction, which can add up to a significant amount due to the sheer size of the network.

GPU kernel 1 performs two functions. Based on the current community status of the network, the assigned GPU thread converts each neighboring node ID in the data structure to its corresponding community ID. The thread also prepares the key for the GPU radix sort in the next step. The GPU radix sort arranges the entire array first in



Community	Nodes
0	0, 2
1	1, 3, 6
2	4, 5
3	7, 8, 9, 10, 11, 12,13

Data structure

Node ID	0	0	0	0	1	1	2	2	2	3	3	3	3	3	3	4	4	4
Neighboring node ID	1	2	3	4	0	3	0	3	5	0	1	2	4	6	11	0	3	5
Outgoing link weight	1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	0	1	1
Incoming link weight	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	1	0	0

GPU kernel 1

Node ID	0	0	0	0	1	1	2	2	2	3	3	3	3	3	3	4	4	4
Neighboring node ID	**1**	**0**	**1**	**2**	**0**	**1**	**0**	**1**	**2**	**0**	**1**	**0**	**2**	**1**	**3**	**0**	**1**	**2**
Outgoing link weight	1	1	1	1	0	1	0	0	0	0	0	1	0	1	1	0	1	1
Incoming link weight	0	0	0	0	1	0	1	1	1	1	1	0	1	0	0	1	0	0

GPU radix sort

Node ID	**0**	**0**	**0**	**0**	**1**	**1**	**2**	**2**	**2**	**3**	**3**	**3**	**3**	**3**	**3**	**4**	**4**	**4**
Neighboring node ID	**0**	**1**	**1**	**2**	**0**	**1**	**0**	**1**	**2**	**0**	**0**	**1**	**1**	**2**	**3**	**0**	**1**	**2**
Outgoing link weight	1	1	1	1	0	1	0	0	0	0	1	0	1	0	1	0	1	1
Incoming link weight	0	0	0	0	1	0	1	1	1	1	0	1	0	1	0	1	0	0

GPU kernel 2

Node ID	0	**0**	0		1	1	2	2	2	**3**	**3**	3	3			4	4	4
Neighboring node ID	0	**1**	2		0	1	0	1	2	**0**	**1**	2	3			0	1	2
Outgoing link weight	1	**2**	1		0	1	0	0	0	**1**	**1**	0	1			0	1	1
Incoming link weight	0	**0**	0		1	0	1	1	1	**1**	**1**	1	0			1	0	0

Fig. 3. Example to illustrate mapping of Find Neighboring Communities to GPU

order of increasing node ID and then in order of increasing neighboring community ID for array elements with the same node ID. The radix sort in the Thrust library is used in this paper. With the sorted array, each node is being assigned a GPU thread in GPU kernel 2. The thread goes down the array elements belonging to the node and sums up the weights for adjacent elements with the same neighboring community ID to give the final output of FNC. As can be seen in Fig. 3, node 0 has outgoing links to communities 0, 1, and 2 and the weights of the links are as shown.

5.2 Mapping of Find Best Move to GPU

The second and third levels of parallelism are considered when mapping Find Best Move (FBM) to GPU. As described in Section 4, the second level of parallelism involves dividing the nodes in the network into batches, with the nodes in each batch being processed in parallel. In the GPU implementation, the GPU kernel for FBM assigns a number of threads to each node in a batch. The threads handle the third level of parallelism by computing the gain in modularity of inserting the node into each of its neighboring communities in batches. The first thread of the assigned threads is also responsible for all computations in the second level of parallelism, such as determining which neighboring community offers the maximum positive gain in modularity and moving the node into the community.

5.3 Multi-core with Multi-GPU Implementation

The highest level of parallelism is implemented by using a multi-core with multi-GPU architecture as illustrated in Fig. 4 for a four-partition example. A simple partitioning scheme, which divides the nodes in the network evenly between the sub-networks (SNs) based on their node IDs, is used. The links that join nodes residing in different sub-networks are set aside. The Louvain method is then applied to solve the community detection problem in each sub-network in parallel. The Louvain method incorporates the second and third levels of parallelism as described in the previous two sections. It is to be highlighted that since the profiling results show that most of the computation time is spent in the first pass of the Louvain method, only the FNC and FBM in the first pass is offloaded to the GPU. The parts of the Louvain method that have been offloaded to the GPU are implemented using a multi-GPU architecture, while the serial parts of the Louvain method are implemented as a multi-core architecture. The results obtained for the individual sub-networks are then combined with the removed links into a single network. The serial version of the Louvain method is applied once more on this combined network to obtain the final community results.

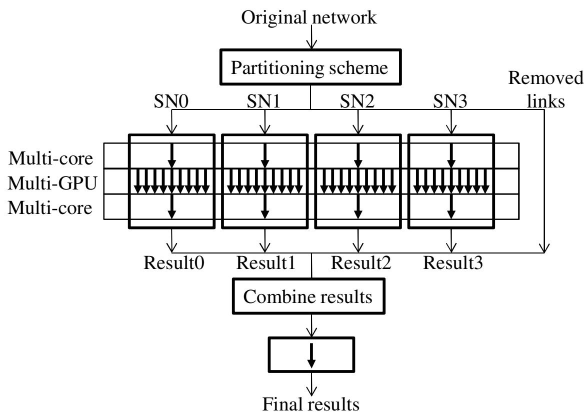


Fig. 4. Illustration of multi-core with multi-GPU implementation

6 Evaluation

The performance of the hierarchical parallel algorithm for community detection proposed in this paper is evaluated using the networks in Table 1. The considered networks include large web-based networks and popular social networks. The results reported in this section were obtained on a server with Intel Xeon E5405 2 GHz processor and four NVIDIA Fermi C2070 GPUs. All executables have been obtained using the '-O3' compiler optimization option. Due to memory limitations of the server, the original Louvain method was not able to process the larger web-based networks, namely uk-2005 and webbase-2001, as a whole. As such, a sub-network of the original network was generated using random sampling.

Table 1. Details of the networks used in the evaluation

Network	Description	# Vertices (M)	# Links (M)	Degree	Reference
uk-2005	Web network (.uk domain)	15.99	287.20	17.96	[9]
webbase-2001	Web network	19.97	138.07	6.91	[10]
twitter	Twitter user follow links	3.26	13.13	4.03	Twitter API
flickr	Flickr follow links	2.30	33.14	14.39	[8]
livejournal	LiveJournal social network	5.20	76.94	14.79	[7]
orkut	Orkut social network	3.07	223.53	72.75	[7]
facebook	Facebook social network	2.94	41.92	14.26	[6]

6.1 GPU Thread Configuration

The performance of the GPU is generally susceptible to its thread configuration. This section seeks to find the optimal thread configuration for the single-GPU implementation of the hierarchical parallel algorithm. This means that only the second and third levels of parallelism are considered. The computation times for the different thread configurations obtained on the uk-2005 network are shown in Table 2.

Table 2. Computation times (in seconds) for different GPU thread configurations on uk-2005

# Blocks per SM	2		4		6		8	
# Threads per block	# Threads per node		# Threads per node		# Threads per node		# Threads per node	
	16	32	16	32	16	32	16	32
256	181.59	243.99	134.71	182.32	122.22	146.72	118.4	132.66
512	133.8	183.02	118.02	134.07	110.89	122.72	107.17	119.02
768	123.6	146.9	112.02	124.37	108.06	118.41	**106.94**	113.62

Table 3. Computation times (in seconds) for uk-2005 using different GPU thread configurations but fixing the number of nodes that is processed in parallel per SM

# Nodes processed in parallel per SM	# Threads per node				Configuration	
	4	8	16	32	# Blocks per SM	# Nodes per block
32	188.82	192.74	179.81	182.32	4	8
32	188.54	186.14	181.59	183.02	2	16
48	150.76	146.96	143.84	146.72	6	8
48	151.34	148.28	146.43	146.9	2	24
64	137.32	134.69	131.86	132.66	8	8
64	137.57	135.47	134.71	134.07	4	16
96	125.76	123.89	122.22	122.72	6	16
96	124.42	124.03	121.73	124.37	4	24

In Table 2, by fixing the number of blocks per SM and the number of threads in each block and then assigning different number of threads to each node, the degrees of parallelism in the second and third levels can be controlled. Assigning a larger number of threads to each node would increase the degree of parallelism in the third level since there are more threads to compute the gain in modularity of inserting the node into its neighboring communities. However, this would lead to a corresponding decrease in the degree of parallelism in the second level with less nodes being able to be processed in parallel. It is clear from Table 2 that assigning 16 threads to each node has a performance advantage over assigning 32 threads to each node. This result is likely due to the fact that the uk-2005 network has a degree of 17.96 and some of the 32 threads assigned to each node would be idling.

To study how the degrees of parallelism in the second and third levels affect computational performance, another set of experiments, whose results are tabulated in Table 3, is performed. In these experiments, the number of nodes that are processed in parallel per SM, which controls the degree of parallelism in the second level, is fixed at 32, 48, 64, and 96. The number of threads assigned to each node, which determines the degree of parallelism in the third level, is set at 4, 8, 16, and 32. It can be seen in Table 3 that by fixing the number of nodes that are processed in parallel per SM, the number of threads assigned to each node has a small, but not negligible, effect on computation times. In all settings, assigning 16 threads to each node resulted in the lowest computation times. The results in Table 3 also show that the GPU configuration, i.e. the number of blocks per SM and the number of nodes per block, has very little effect on computational performance as well. The main parameter that affects performance is the number of nodes that are processed in parallel per SM. The larger the number of nodes processed in parallel per SM, the better the performance.

The best configuration is highlighted in bold in Table 2 and is used to obtain the rest of the results in this paper.

6.2 Comparison of Computation Times

This section assesses the performance, in terms of computation time, of the parallelized Louvain method proposed in this paper. The computation times for three versions of the Louvain method are compared in Table 4. In Table 4, Louvain is the original Louvain method [4], SingleGPU utilizes the second and third levels of parallelism, and MultiGPU considers the highest level of parallelism as well by splitting the original network into four sub-networks. The amount of computational speedups SingleGPU has over Louvain and MultiGPU has over both Louvain and SingleGPU are also shown in the table.

The results in Table 4 show that SingleGPU offers varying degrees (3x on average) of speedup over Louvain. The highest speedups are achieved on the three largest graphs, i.e. uk-2005, webbase-2001, and orkut. On average, MultiGPU offers another 2x speedup over SingleGPU. MultiGPU did not perform as well on facebook, only achieving a speedup of 1.27x over SingleGPU. A detailed examination revealed that MultiGPU suffered a load balancing problem. For facebook, although the nodes in the network were divided evenly between the sub-networks, one of the sub-networks had a disproportionate number of links, resulting in an uneven distribution of the computational load.

Table 4. Computation times (in seconds) for three versions of the Louvain method

Network	Louvain	SingleGPU		MultiGPU		
		Time	Speedup over Louvain	Time	Speedup over Louvain	Speedup over SingleGPU
uk-2005	497.12	109.94	4.52	56.15	8.85	1.96
webbase-2001	419.61	105.82	3.97	52.46	8.00	2.02
twitter	130.54	73.03	1.79	20.97	6.23	3.48
flickr	113.67	66.04	1.72	27.36	4.15	2.41
livejournal	273.1	145.72	1.87	83.84	3.26	1.74
orkut	1683.3	338.13	4.98	100.45	16.76	3.37
facebook	165.76	51.55	3.22	40.55	4.09	1.27

6.3 Comparison of Modularity Values

While the hierarchical parallel algorithm proposed in this paper offers considerable speedups over the original Louvain method, two sources of inaccuracy have been introduced in the parallel versions. In SingleGPU, the second level of parallelism assumes that the computations for visiting any two nodes in a batch are independent, which may not be the case since one of the nodes may be in the neighboring community of the other node. In addition, inaccuracy is introduced when MultiGPU partitions a network into sub-networks, solves the community detection problem in each sub-network independently, and then combines the results. To study the extent to which the inaccuracies affect community detection results, a comparison of the modularity values obtained by the three versions of the Louvain method is provided in Table 5. The best result for each network is highlighted in bold. The percentage differences in results between the two proposed versions and Louvain are also given.

Table 5. Modularity values for three versions of the Louvain method

Network	Louvain	SingleGPU		MultiGPU	
		Q	% difference	Q	% difference
uk-2005	**0.998**	**0.998**	0	**0.998**	0
webbase-2001	**0.998**	**0.998**	0	**0.998**	0
twitter	**0.606**	0.598	-1.32	0.583	-3.8
flickr	0.655	**0.665**	1.53	0.641	-2.14
livejournal	0.734	**0.756**	3	0.701	-4.50
orkut	0.661	**0.682**	3.18	0.600	-9.23
facebook	0.716	**0.730**	1.96	0.709	-0.98

From Table 5, it can be observed that with the exception of the twitter network, the modularity values obtained by SingleGPU are equal, if not better, than those obtained by the original Louvain method. It is clear from these empirical results that the inaccuracy introduced by assuming that the computations for visiting any two nodes in a batch are independent does not have a negative impact on the modularity values.

However, from the modularity values of the solutions obtained by MultiGPU, it can be observed that the inaccuracy introduced by solving the sub-networks rather than solving the original network as a whole has caused an average of 3% degradation in the modularity values. Given the speedups it offers, MultiGPU would still be useful in providing a quick and approximate community detection solution.

7 Conclusions

This paper represents the first attempt to accelerate the Louvain method, or modularity-based methods in general, on the GPU platform. Benchmarking results on several large web-based networks and popular social networks show that on top of offering speedups, the single-GPU version of the proposed hierarchical parallel algorithm is able to find better quality communities. The multi-GPU version provides additional speedups over the single-GPU version but with a slight degradation in community quality.

A future work would be to design a more effective method to partition the network into sub-networks at the highest level of parallelism. The method should address two problems with the current design. Firstly, the computational load for each sub-network needs to be more balanced. Secondly, the degradation in community quality needs to be minimized. The former could be tackled by partitioning the network such that the sub-networks have approximately equal number of links. The latter is more challenging as it requires the network partitioning problem to be treated as a community detection problem so that nodes that would eventually be in the same community are placed in the same sub-network. A bigger challenge lies in integrating the two solutions as their objectives are potentially conflicting in nature.

References

1. Fortunato, S.: Community Detection in Graphs. Physics Reports (2009)
2. Flake, G.W., Lawrence, S., Giles, C.L., Coetzee, F.M.: Self-Organization and Identification of Web Communities. IEEE Computer 35(3), 66–71 (2002)
3. Green, D.M., Werkman, M., Munro, L.A., Kao, R.R., Kiss, I.Z., Danon, L.: Tools to Study Trends in Community Structure: Application to Fish and Livestock Trading Networks. Preventive Veterinary Medicine 99, 225–228 (2011)
4. Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast Unfolding of Communities in Large Networks. Journal of Statistical Mechanics: Theory and Experiment, P10008 (2008)
5. NVIDIA Kepler GK110 Architecture Whitepaper, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`
6. Wilson, C., Boe, B., Sala, A., Puttaswamy, K.P.N., Zhao, B.Y.: User Interactions in Social Networks and Their Implications. In: 2009 EuroSys Conference (2009)
7. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and Analysis of Online Social Networks. In: 5th ACM/Usenix Internet Measurement Conference, IMC (2007)

8. Cha, M., Mislove, A., Gummadi, K.P.: A Measurement-Driven Analysis of Information Propagation in the Flickr Social Network. In: 18th International World Wide Web Conference (2009)
9. Laboratory for Web Algorithmics, http://law.dsi.unimi.it/
10. Stanford WebBase Project, http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/
11. Girvan, M., Newman, M.E.J.: Community Structure in Social and Biological Networks. National Academy of Sciences 99(12), 7821–7826 (2002)
12. Newman, M.E.J., Girvan, M.: Finding and Evaluating Community Structure in Networks. Physical Review E 69(2) (2004)
13. Zhang, Y., Wang, J., Wang, Y., Zhou, L.: Parallel Community Detection on Large Networks with Propinquity Dynamics. In: 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 997–1006 (2009)
14. Eriksen, K.A., Simonsen, I., Maslov, S., Sneppen, K.: Modularity and Extreme Edges of the Internet. Physical Review Letters 90(14) (2003)
15. Pujol, J.M., Erramilli, V., Rodriguez, P.: Divide and Conquer: Partitioning Online Social Networks (2009), http://arxiv.org/abs/0905.4918v1
16. Haynes, J., Perisic, I.: Mapping Search Relevance to Social Networks. In: 3rd Workshop on Social Network Mining and Analysis (2010)
17. Hui, P., Sastry, N.: Real World Routing Using Virtual World Information. In: International Conference on Computational Science and Engineering, vol. 4, pp. 1103–1108 (2009)
18. Soman, J., Narang, A.: Fast Community Detection Algorithm with GPUs and Multicore Architectures. In: IEEE International Parallel & Distributed Processing Symposium, pp. 568–579 (2011)