# Highly Scalable Community Detection Using a GPU

Md. Naim*, Fredrik Manne*, Mahantesh Halappanavar†, and Antonio Tumeo†

Community detection typically consists of discovering sets of nodes such that the number of edges between different sets is relatively low compared to the number of edges within sets. Although there is no standard mathematical definition for what makes up a community, the modularity metric proposed by Girvan and Newman is often used. This is a measurement of the density of links within communities as compared to the density of inter-community links.

We present a new highly scalable GPU implementation of the Louvain method for community detection. The Louvain methods iterates over two alternating stages, the modularity optimization phase and the community aggregation phase. In the first phase vertices can move between communities if this increases the local modularity. When no further modularity gain can be achieved in this way, the vertices in each community are agglomerated into a new super vertex in the aggregation phase. This condensed graph then become the input to the next modularity optimization phase and so on. The final output of the method will thus be a cluster hierarchy. Note that at the start of the algorithm each vertex is a community by itself.

There has been several previous efforts to parallelize the Louvain method. Wickramaarachchi et al. gave an MPI implementation with a speedup of up to 5 using 128 processors [5]. Ovelgönne presented a Hadoop implementation [3]. Cheong et al. gave a GPU implementation with a speedup up to 5 on a single GPU and 17 on a multi-GPU computer [1]. Lu et al. presented an OpenMP implementation, reporting speedups up to 16 using 32 threads [2]. Xinyu et al. recently presented an implementation using a 8,192 nodes Blue Gene/Q and a 1024 node P7-IH super computer [4]. They report processing rates of up to 1.89 giga TEPS on graphs containing up to 138 billion edges, where TEPS is the number of edges processed per second in the first modularity phase.

Our algorithm is based on the one presented by Lu et al. We therefore start by reviewing this. When processing a node $v$ in the modularity optimization phase one must determine the accumulated weight of the edges that $v$ has to each of its neighboring communities. To do this one uses a hash map where the weight of each incident edge is inserted using the current community ID of the corresponding neighbor as key. Once this is done it is possible to determine which community will give the largest modularity gain if $v$ was to move to it. In the sequential algorithm one iterates over the vertices, and for each vertex $v$ computes to which (if any) community $v$ should move to. If $v$ is to move then its community ID is updated accordingly. This strategy might cause problems in a parallel implementation as several vertices might be moving at the same time thus making the computation of the modularity gains imprecise. Lu et al. therefore suggested that vertices should compute and update their community IDs in separate stages. Thus in the first stage every vertex computes to which community it should belong, but without changing its community ID. Only after every vertex has performed this computation will all vertices concurrently move to their new communities. In this way the difference between the sequential and the parallel algorithm is similar to that between the Gauss-Seidel and Jacobi iterative methods for solving systems of linear equations.

In the parallel algorithm the vertices are divided between the threads who sequentially compute to which community each of its allocated vertices should move. Only after a global barrier are the community IDs updated. This process is repeated until the change in modularity is sufficiently small. In the community aggregation phase the communities are distributed between the threads who compute the structure of the new graph to be used as input in the next round of modularity optimization.

It should be noted with that with this strategy there is always only one thread that is responsible for each vertex in the modularity optimization. Similarly, there is always only one thread responsible for each community in the aggregation phase. Thus if vertices have very different degrees or if the number of vertices in each community differ substantially this scheme can lead to problems with uneven load balance.

We next outline how we have adapted the algorithm to run on a GPU. On a GPU the threads are first grouped into blocks and each block is further divided into warps, where the threads in each warp operate in SIMD. On our GPU each warp consists of 32 threads. The main problem we have to consider is how to allocate tasks to threads so as to achieve an even load balance.

We first consider the modularity optimization phase.

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: {fredrikm,md.naim}@ii.uib.no

†Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O.Box 999, MSIN J4-30, Richland, WA 99352, USA. Email: {Mahantesh.Halappanavar,Antonio.Tumeo}@pnnl.gov

| Graph | #V | #E | Time$_{seq}$ | Time$_{OMP}$ | Modularity$_{OMP}$ | Time$_{GPU}$ | Modularity$_{GPU}$ | TEPS | Speedup$_{Seq}$ | Speedup$_{OMP}$ | Ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CNR | 325557 | 2738969 | 4.3 | 0.8 | 0.912608 | 0.37 | 0.912447 | 3.43E+007 | 11.62 | 2.16 | 1.000 |
| coPapersDBLP | 540486 | 15245729 | 7.7 | 3.7 | 0.858088 | 0.79 | 0.857094 | 2.72E+007 | 9.75 | 4.68 | 0.999 |
| channel | 4802000 | 42681372 | 30.9 | 21.2 | 0.933388 | 9.22 | 0.927304 | 1.38E+008 | 3.35 | 2.30 | 0.993 |
| europe_osm | 50912018 | 54054660 | N/A | 63.4 | 0.994996 | 32.13 | 0.998677 | 9.20E+006 | N/A | 1.97 | 1.004 |
| uk-2002 | 18520486 | 261787258 | 335.9 | 210.3 | 0.989569 | 8.27 | 0.989351 | 5.10E+007 | 40.62 | 25.43 | 1.000 |
| rgg_n_2_24_s0 | 16777216 | 132557200 | 111 | 34.2 | 0.992698 | 7.14 | 0.992521 | 3.28E+007 | 15.55 | 4.79 | 1.000 |
| Soc-LiveJournal1 | 4847571 | 68475391 | 182.7 | 67.05 | 0.751404 | 9.15 | 0.753148 | 1.80E+007 | 19.97 | 7.33 | 1.002 |

Table 1: Comparison with OMP implementation

If we allocate vertices to different threads within the same warp, then the thread that has to handle the most neighbors will determine the time spent. Similarly, if we allocate vertices to entire warps, then the most heavily loaded warp will determine when a particular block finishes. To achieve an even load balance we chose to distribute the vertices based on their vertex degrees. Thus we first place all the vertices in bins depending on their degrees. We then process the vertices in each bin before proceeding with the next one. In this way we can update the community ID of all vertices in one bin before moving on to the vertices in the next bin. The number of threads assigned to each vertex depends on the bin it belongs to. For low to medium degree vertices the number of threads assigned varies from a fraction of a warp up to an entire warp, but remains consistent for all warps in a thread block. The threads assigned to one vertex uses a common area in the shared memory for hashing. The threads participate in reading neighbor information for the current vertex and putting this into the hash map. To avoid collisions we use atomic operations to protect the actual updating of the hash table. When all neighbors have been processed the threads perform a reduction operation to determine which community the current vertex should belong to. For vertices of high degree we use an entire thread block for processing each vertex. Depending on the degree of these vertices we either use shared or global memory for storing the hash table.

In the community aggregation phase the vertices are first grouped according to which community they belong to. Then depending on the combined size of the neighborhood of the vertices in each community, either a block or a warp is assigned to perform the aggregation. This is done by hashing the neighbors of each vertex in a community into a common hash map. In this way we can compute the new neighborhood of the vertex that is to replace the current community. We note that this phase is not load balanced as thoroughly as the modularity optimization, but on the other hand the modularity optimization is run many times for each time that the aggregation phase is run.

Our experiments are run on an NVidia Tesla K40m GPU with 12 GB of memory, 2880 cores, running at a 745 MHz. In Table 1 we present results from seven graphs that were also used by Lu et al. For each graph we first list the number of vertices, the number of edges, the sequential running time, the running time of the OpenMP code using 8 threads, and the final modularity as reported in [2]. Next, we give the running time of our GPU code, achieved modularity, and the number of edges traversed per second in the first modularity optimization stage. Finally, we give the speedup of the GPU code relative to the sequential code and to the OpenMP code running on 8 threads, and also the ratio between the modularities. We do not compare against more than 8 threads for the OpenMP implementation as these were the numbers presented in [2] and there were only marginal improvements for higher number of threads.

As can be seen from the results the speedup of the GPU implementation compared to the OpenMP one ranges from approximately 2 up to more than 25. We also note that the modularity is very similar for both implementations. We note that our maximal TEPS rate is 7.3% of that achieved by [4] on the 8,192 nodes Blue Gene/Q.

## References

[1] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. Hierarchical parallel algorithm for modularity-based community detection using gpus. In *Euro-Par 2013 Parallel Processing*, pages 775–787. Springer, 2013.

[2] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[3] Michael Ovelgönne. Distributed community detection in web-scale networks. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 66–73. ACM, 2013.

[4] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. Scalable community detection with the louvain algorithm. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 28–37. IEEE, 2015.

[5] Charith Wickramaarachchi, Marc Frincu, Patrick Small, and Viktor K Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.