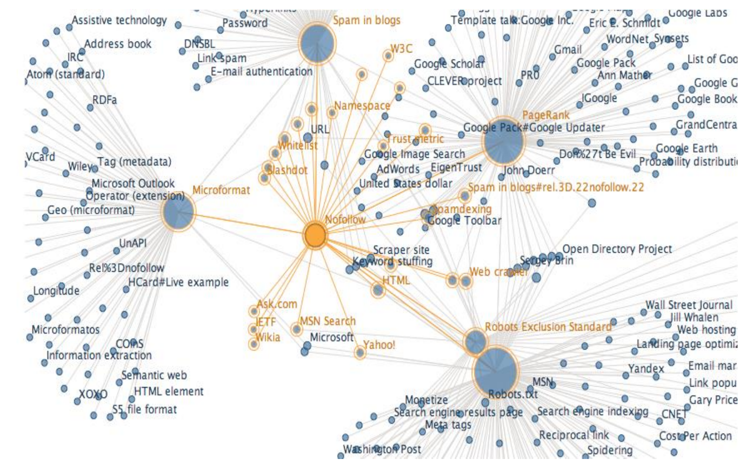
Heterogeneous Computing for Graph Algorithms

Sathish Vadhiyar
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India

Department of Computational and Data Sciences

Introduction

- Graph processing has been prevalent
- Real world graphs large in size
- Processing such real world graphs requires effective harnessing of multiple nodes of CPU and GPU devices



GPUs

- Graphical Processing Units (GPUs)
- A single CPU can consist of 2, 4, 8 or 12 cores
- GPUs consist of a large number of light-weight cores
- Typically GPU and CPU coexist in a heterogeneous setting
- "Less" computationally intensive part runs on CPU (coarse-grained parallelism), and more intensive parts run on GPU (fine-grained parallelism)

Challenges

- Irregular memory access, hence poor locality.
- Poor computation-to-communication ratio.
- Varying parallelism while execution.
- Frequent need of synchronization.
- Load Balancing across computing units.
- **Most important:** To be able to use all the heterogeneous resources

Divide and Conquer

- Partition and run the algorithm independently on the devices

- Three cases
  - *Case 1:* Control the independent algorithmic computations – MST
  - *Case 2:* Split into batches and different pipeline stages on the devices – Betweenness centrality
  - *Case 3:* Let loose and correct – Community Detection

Case 1: Minimum Spanning Tree (MST)

- MST one of the important graph applications
- Large scale MST requires multiple nodes with distributed memory parallelism

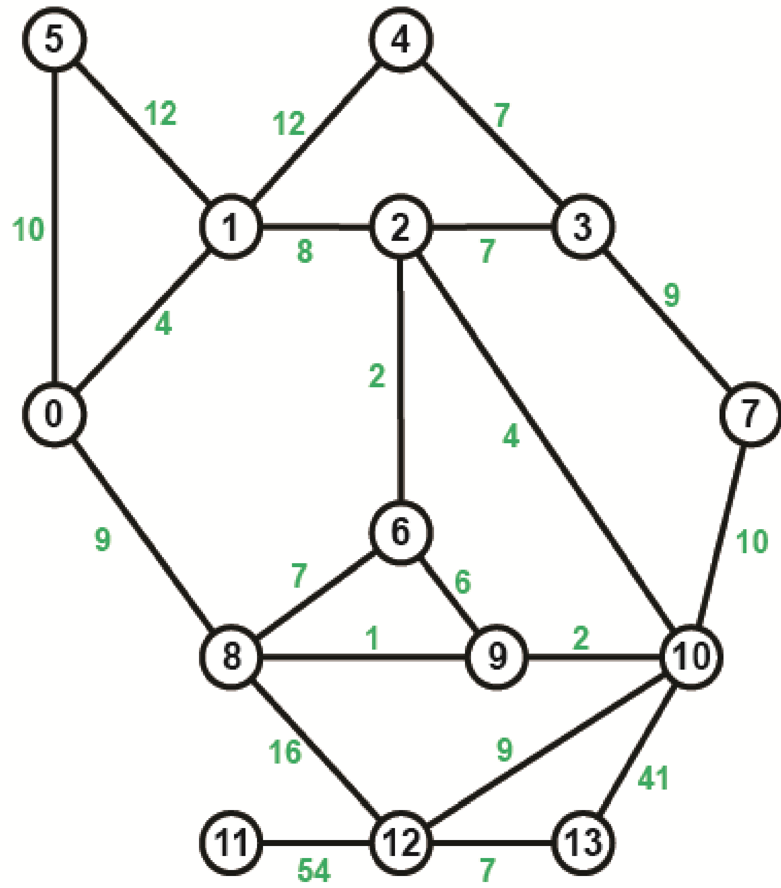
Multi-node multi-device MST

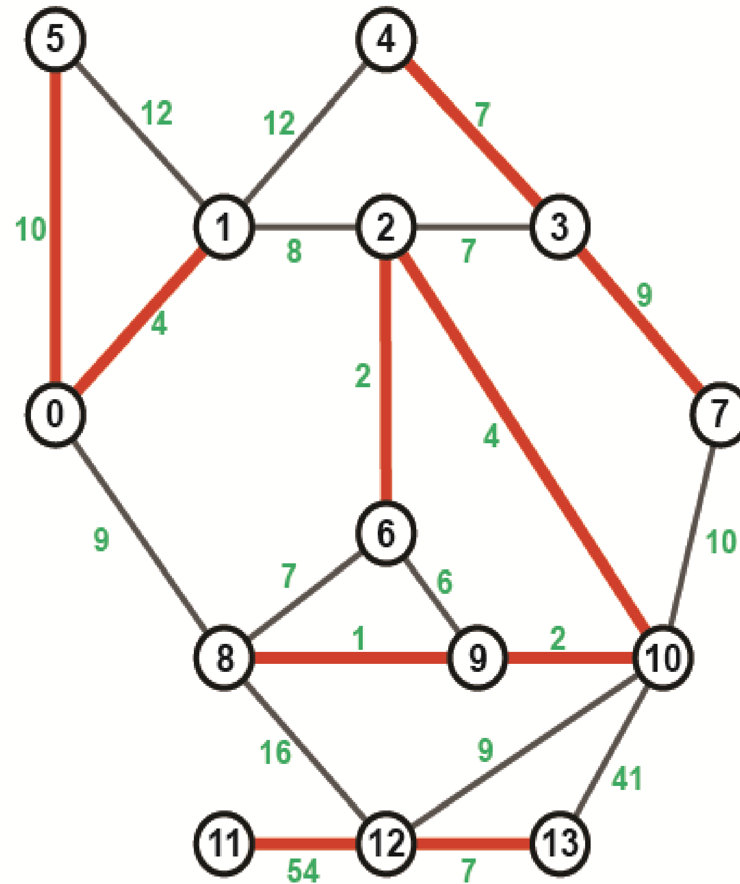
- We propose a multi-node multi-device algorithm following a divide-and-conquer paradigm
- Graph is partitioned across multiple nodes and further across multiple devices
- Boruvka's MST invoked independently on each partition/device
- Results merged using a novel hierarchical merging algorithm

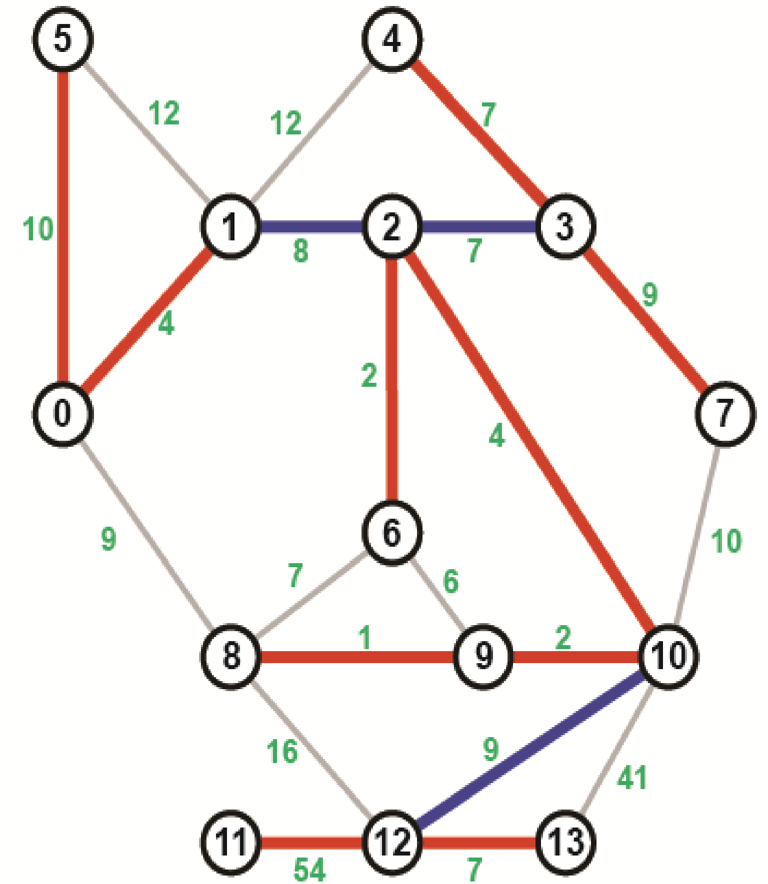
Boruvka's MST

- Iteratively finds lightest edges from a component and merges two components connected by a lightest edge – called edge contraction
- Process repeated until a single component formed
- Edges contracted across all the iterations form the minimum spanning tree

Boruvka’s MST







Hybrid CPU-GPU MST

- Consists of:
  - Partitioning
  - Independent computations:
    - Run the Bouruvka's algorithm on each device
    - But **don't run** the algorithm on the border vertices
  - Merging
  - Post processing

Implementation: HyPar Divide-and-conquer API

Function	Remarks
partGraph	Partitions the graph into number of processing units.
indComp	Performs independent computations of a graph kernel, given by an *appName*, on each partitions independently with **excpCond** and returns the result.
mergeParts	Merges the results from the independent computations on the devices and communicates ghost vertices.
postProcess	Performs postprocessing by executing the kernel given by *postProcessKernelName* with updated graph as the input.

HyPar Runtime Optimizations

1. **Ratio for Graph Partitioning:**
   - To find the ratio for partitioning we use a heuristic approach. We choose 5-10 random induced subgraphs with 5% of the total number of nodes and run the application on both the CPU and GPU devices simultaneously to find the partitioning ratio.
2. **Threshold for Independent Computation**
   - While performing independent computation in each partition, the amount of parallelism may drop significantly after few iterations.
   - HyPar automatically find the threshold by observing trend in execution times, and switches to the merging step at the threshold
3. **Recursive Invocation of Partitioning-Independent Computations-Merging**
   - After mergeStep in many applications the remaining graph size may be large.
   - HyPar framework again partitions the reduced graphs using the same partitioning ratio, followed by invocation of indComp and mergeStep if the size of the graph is more than a threshold

Case 2: Betweenness Centrality

- Betweenness Centrality is a shortest path metric used to give a score to each vertex in a graph or network based on how many shortest paths it lies on.
- Definition:-
  - For a graph $G = (V,E)$, where V is the set of vertices and E, the set of edges. Let $\sigma_{st}(v)$ denotes the number of shortest path from vertex *'s'* to vertex *'t'*, where $s \neq t$, passing through vertex v.
  - Based on above, we find $\delta_{st}(v) = \sigma_{st}(v) / \sigma_{st}$, where $\delta_{st}(v)$ denotes the pair-wise dependency between of the pair *'s'* and *'t'* on *'v.'*
  - The Betweenness Centrality score of the vertex is given by

$$BC(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$$

- One way to find BC for all vertices is to perform an APSP and aggregate the pair-wise dependencies for all vertices. Its costly, $\boldsymbol{O(n^3)}$ and infeasible for larger graphs.

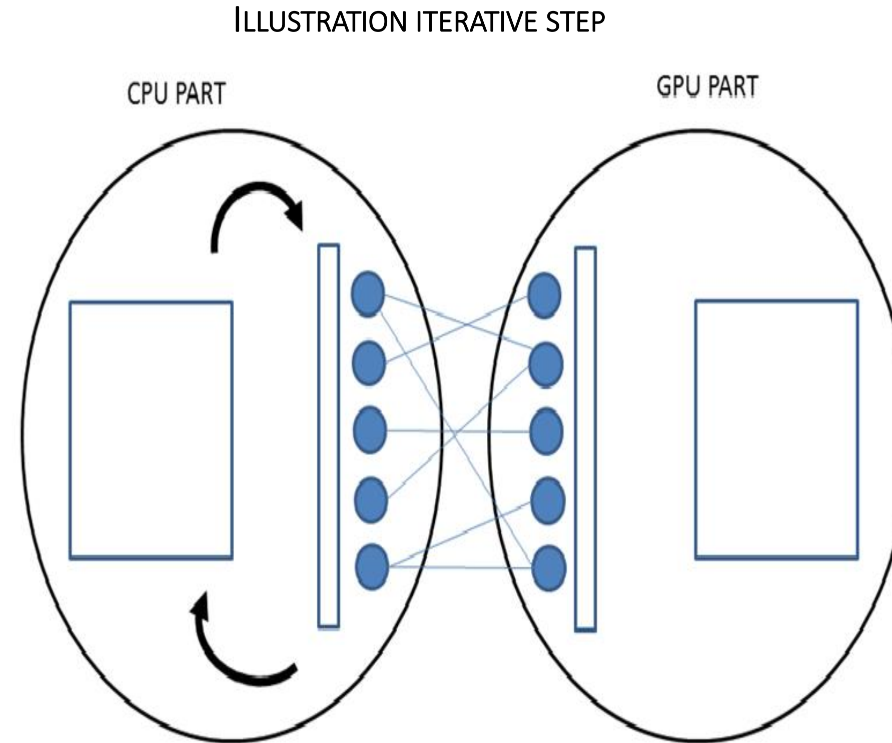
Brandes Algorithm for Betweenness Centrality

- The algorithm by Brandes [JMS-2001] consists of two phases: a **forward** and a **backward** phase.
- The forward phase consists of a BFS traversal or SSSP calculation with s as the source. For each vertex the #shortest paths and predecessor list is calculated.
- The backward phase traverses the vertices in descending order of their distance from s.

Distance calculations in Forward phase.

- In an iteration a source *s* is selected
- Distance values of all nodes in *G* , except *s* are set to ∞, which is set to 0.
- **Step 1**.
  - BFS/SSSP from *s* in *Pr(s)*.
  - $d_C[s, v], \forall v \in \Pr(s)$.
  - **Initial step**.
- **Step 2**.
  - Update $B_{G\text{-}Pr(s)}$ using edge cuts.
  - Using d_C values of $B_{Pr(s)}$, relaxing the values of $B_{G\text{-}Pr(s)}$.
- **Step 3**.
  - Updating the $B_{G\text{-}Pr(s)}$ in the same partition.
  - Using $BM_{G\text{-}Pr(s)}$ for further relaxing the values of $B_{G\text{-}Pr(s)}$.
- **Step 4**.
  - Update $B_{Pr(s)}$ using edge cuts.
  - Using d_C values of $B_{G\text{-}Pr(s)}$, relaxing the d_C values of $B_{Pr(s)}$.
- **Step 5**.
  - Updating the $B_{Pr(s)}$ in the same partition.
  - Using $BM_{Pr(s)}$ for relaxing the d_C values of $B_{Pr(s)}$.

Illustration iterative step

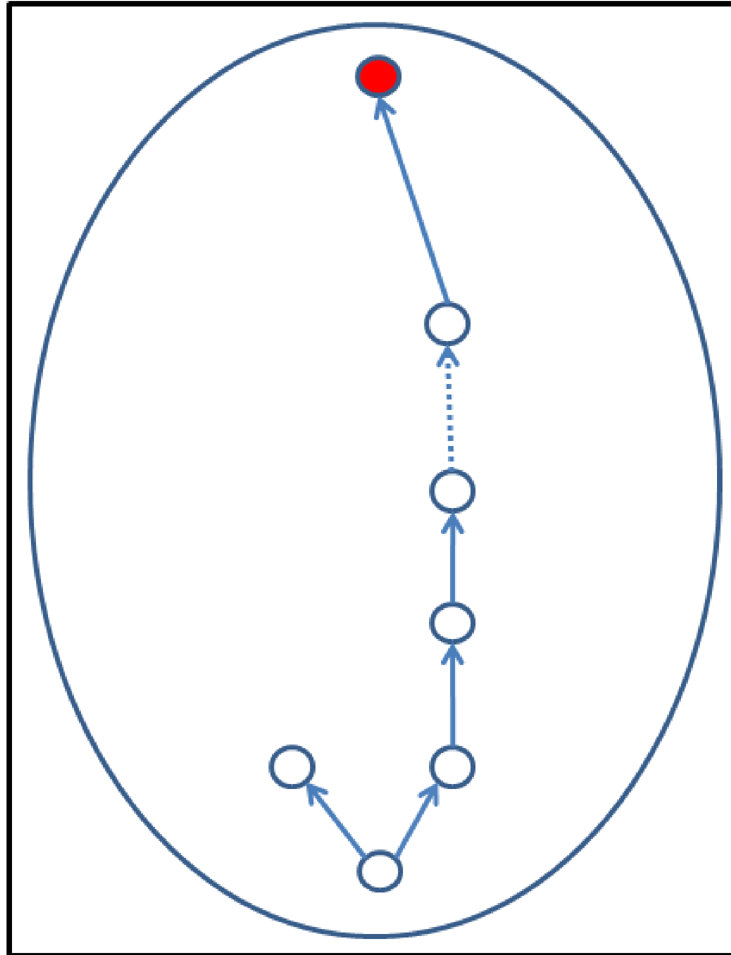


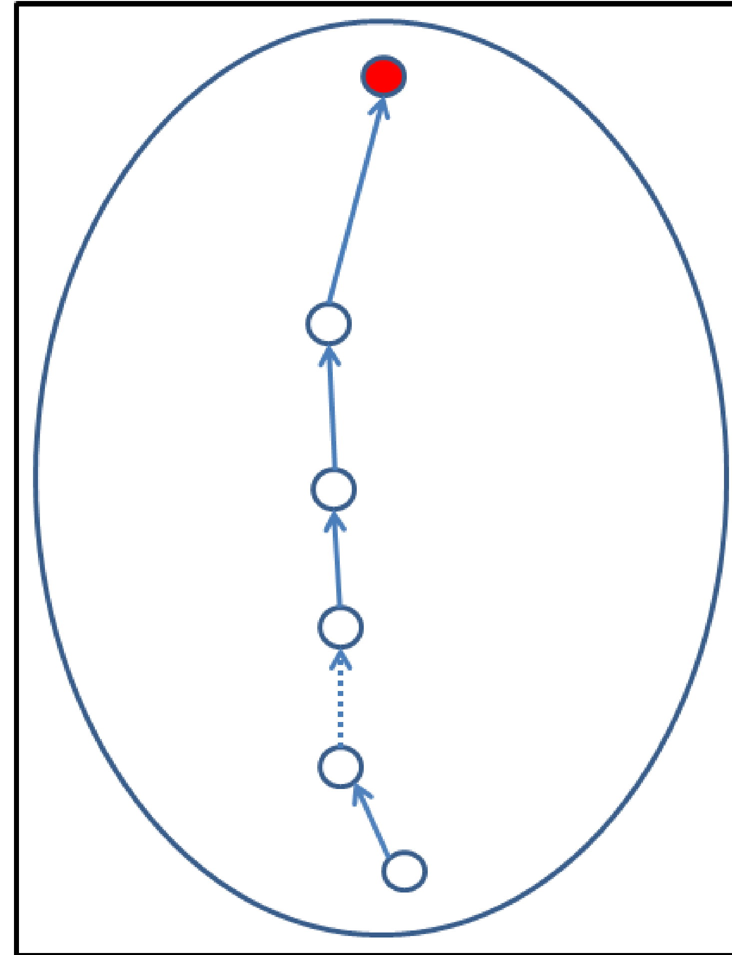
Asynchronous and Hybrid Backward Phase

- The non-zero *edgeσ* characterizes the dependency information.
- The execution is launched simultaneously on CPU and GPU.
- CPU computation threads.
  - In CPU partition.
  - For each *dist* level starting from max till the min.
  - Set ***borderNodeinLevel*** if current level has a border node in GPU.
    - Wait till GPU has completed the current level.

- GPU handler thread.
  - A CPU thread.
  - Invokes the GPU kernel.
  - For each *dist* level starting from max till the min.
  - If ***borderNodeinLevel*** is set them copy the border node values to CPU.
  - If there is a border node in current partition.
    - Wait till CPU has completed the current level.
    - Copy border values from CPU to GPU.

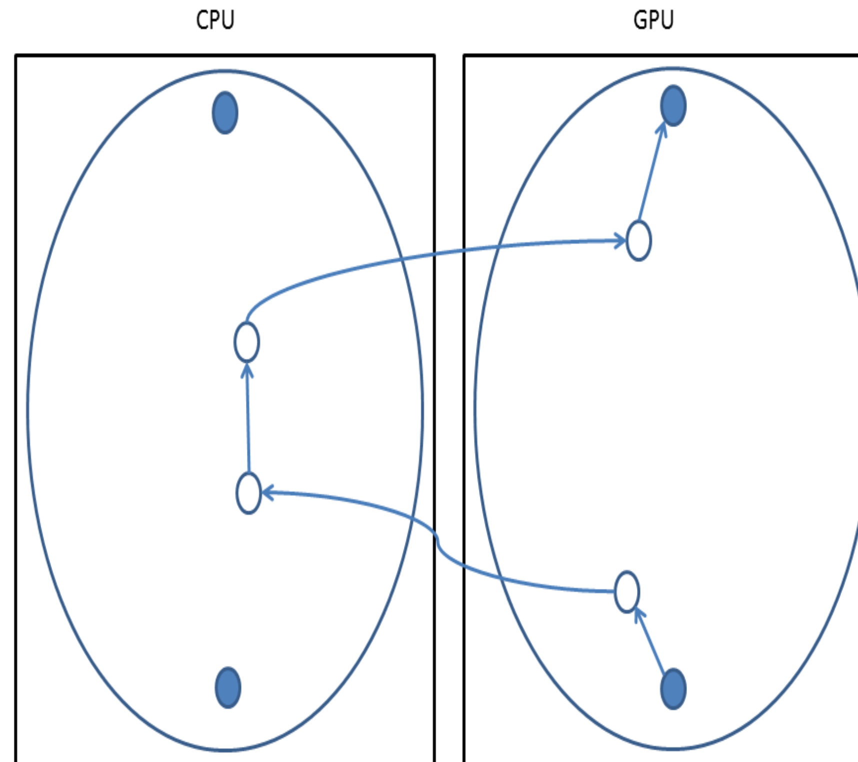
Illustration backward step





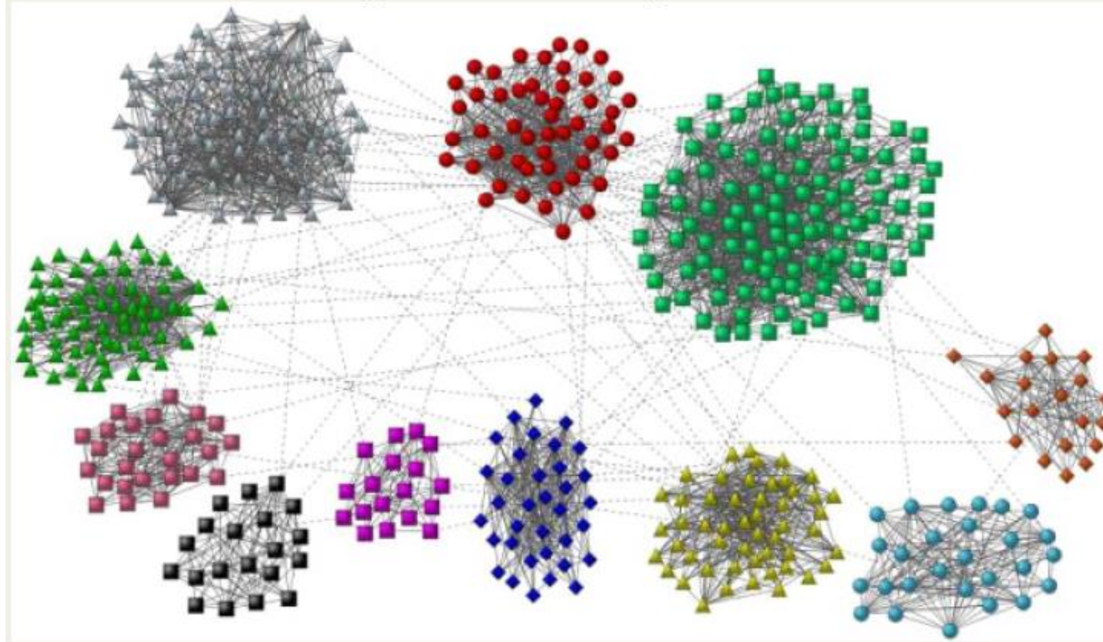
CONTINUED...

- CPU and GPU both traverse asynchronously, until a border node is found.
  - If there is a border node then either processor has to wait for the other to reach current level.
  - Only when required.
- The synchronizations in the backward phase.
  - Depend on the structure of the graphs.
    - Number of border nodes in either partitions.
    - Relation of border nodes among the partitions.
  - Its equal to the number of iterative steps in the forward phase.
- Communication is minimized
  - Only copy the border data structures.
    - delta values.



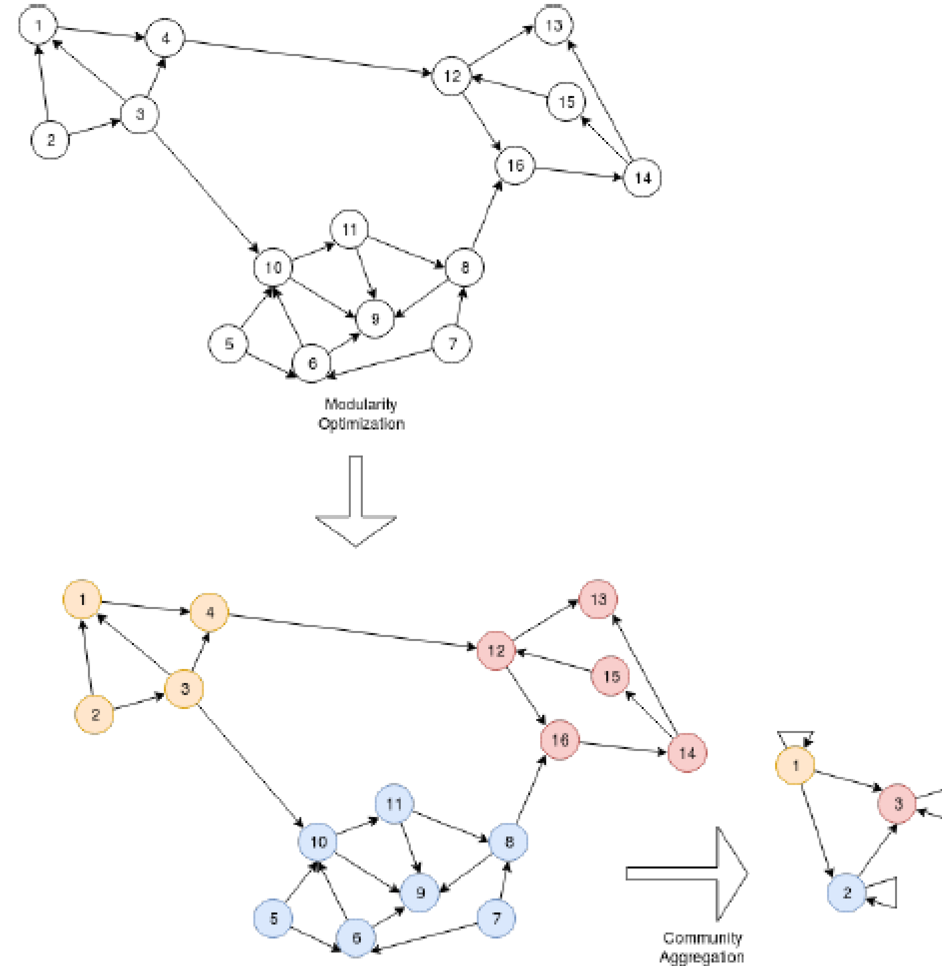
Case 3: Community Detection

- Attempts to identify modules or connected components in a graph
- Used in various fields such as biological science and health care



Hybrid CPU-GPU Algorithm

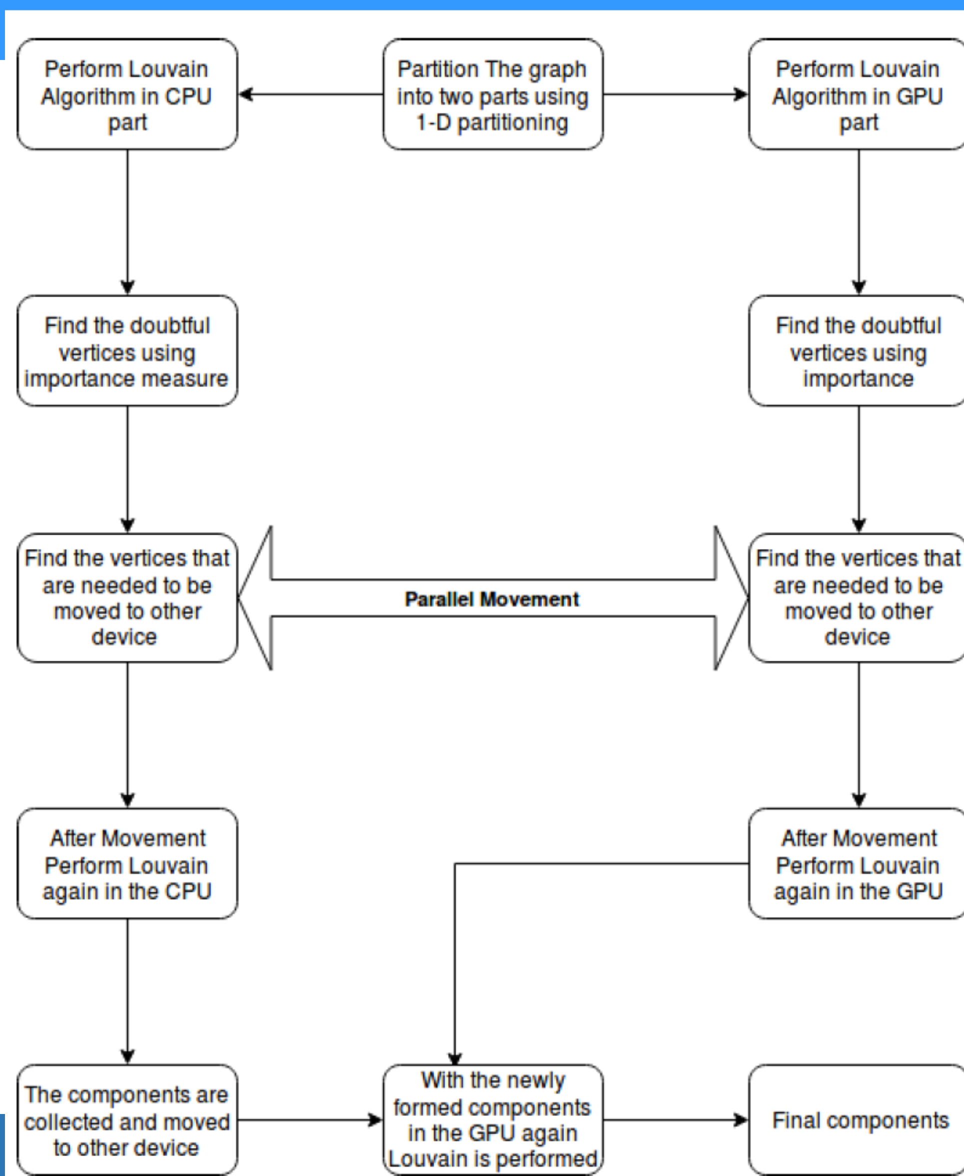
- Once again, partition for CPU and GPU
- Perform independent communities in the devices using a community detection algorithm (e.g., Louvain's)



Hybrid algorithm

- This will form *pseudo communities*
- In the next step, *doubtful vertices* are identified, separated and exchanged
- Independent communities formed again
- Process repeated until components become small

Hybrid Algorithm



Some Lessons

- Take up simple algorithms
  - Simple graph algorithms (e.g., graph coloring), matrix computations
- Try to let it loose across the different devices
- See what needs to be done to get correct answers

Thank You
Questions?