Parallel Graph Algorithms

Sathish Vadhiyar
Graph Traversal

- Graph search plays an important role in analyzing large data sets
- Relationship between data objects represented in the form of graphs
- Breadth first search used in finding shortest path or sets of paths
Parallel BFS

Level-synchronized algorithm

- Proceeds level-by-level starting with the source vertex
- Level of a vertex - its graph distance from the source
- Also, called **frontier-based** algorithm
- The parallel processes process a level, synchronize at the end of the level, before moving to the next level - Bulk Synchronous Parallelism (BSP) model
- How to decompose the graph (vertices, edges and adjacency matrix) among processors?
Distributed BFS with 1D Partitioning

- Each vertex and edges emanating from it are owned by one processor
- 1-D partitioning of the adjacency matrix

Edges emanating from vertex \( v \) is its edge list = list of vertex indices in row \( v \) of adjacency matrix \( A \)
1-D Partitioning

- At each level, each processor owns a set $F$ - set of frontier vertices owned by the processor
- Edge lists of vertices in $F$ are merged to form a set of neighboring vertices, $N$
- Some vertices of $N$ owned by the same processor, while others owned by other processors
- Messages are sent to those processors to add these vertices to their frontier set for the next level
Algorithm 1 Distributed Breadth-First Expansion with 1D Partitioning

1: Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{ where } v_s \text{ is a source} \\ \infty, & \text{otherwise} \end{cases}$

2: for $l = 0$ to $\infty$ do
3: \hspace{1em} $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level $l$
4: \hspace{1em} if $F = \emptyset$ for all processors then
5: \hspace{2em} Terminate main loop
6: \hspace{1em} end if
7: \hspace{1em} $N \leftarrow \{\text{neighbors of vertices in } F \text{ (not necessarily local)}\}$
8: \hspace{1em} for all processors $q$ do
9: \hspace{2em} $N_q \leftarrow \{\text{vertices in } N \text{ owned by processor } q\}$
10: \hspace{2em} Send $N_q$ to processor $q$
11: \hspace{2em} Receive $\bar{N}_q$ from processor $q$
12: \hspace{1em} end for
13: \hspace{1em} $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ \hspace{1em} (The $\bar{N}_q$ may overlap)
14: \hspace{1em} for $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ do
15: \hspace{2em} $L_{v_s}(v) \leftarrow l + 1$
16: \hspace{1em} end for
17: end for

$L_{v_s}(v)$ – level of $v$, i.e., graph distance from source $v_s$
BFS on GPUs

```c
bfs_kernel(int curLevel) {
    v = blockIdx.x * blockDim.x + threadIdx.x;
    if (dist[v] == curLevel) {
        for (sint n ∈ neighbors(v)) {
            if (visited[n] == 0) {
                dist[n] = dist[v] + 1;
                visited[n] = 1;
            }
        }
    }
}
```
BFS on GPUs

- One GPU thread for a vertex
- For each level, a GPU kernel is launched with the number of threads equal to the number of vertices in the graph
- Only those vertices whose assigned vertices are frontiers will become active
- Do we need atomics?
- Severe load imbalance among the threads
- Scope for improvement
MST (Prim’s), SSSP, APSP
Single Source Shortest Path (SSSP)

- Find the shortest distance from a source s to all vertices
- Dijkstra’s algorithm
Single Source Shortest Path (SSSP)

input : a graph \(\text{graph}(V, E)\) with \(N\) vertices in a set \(V\), and \(M\) edges in a set, \(E\), and \(M\) weights, \(w\) of the edges. A source \(s\)

1. forall the \(v \in V\) do
2. \hspace{1em} \(\text{dist}[v] \leftarrow \infty\);
3. end
4. \(\text{dist}[s] \leftarrow 0\);
5. Initialize a priority queue, \(Q\), with all the vertices ordered by the distances;
6. while \(Q \neq \emptyset\) do
7. \hspace{1em} Remove \(u\) with minimum distance from \(Q\);
8. \hspace{1em} forall the \(v \in \text{neighbors}(u)\) do
9. \hspace{2em} if \((\text{dist}[u] + w(u, v)) < \text{dist}[v]\) then
10. \hspace{3em} \text{dist}[v] = \text{dist}[u] + w(u, v);
11. \hspace{3em} Update position of \(v\) in \(Q\);
12. end
13. end
14. end

Fig. 4.21: Sequential Dijkstra’s SSSP Algorithm
Single Source Shortest Path (SSSP)

- The operation of updating the distances of neighbors using the minimum distance of a vertex – relaxation

- Parallelization
  - Vertices distributed across processors
  - Each processor owns a set of vertices and their outgoing edges
  - Priority queue distributed – each processor updates only its vertices in the priority queue
Parallel SSSP Steps

- In each iteration:
  - Minimum of all priority queues found using reduction
  - Processor with the lowest rank removes vertex with the minimum distance
Parallel SSSP Steps

- Performs distributed edge relaxations
  - Processor communicates distance of u to processors that owns u’s neighbors
  - Processors update the tentative distances of neighbors and update their positions in local priority queue

- Disadvantages?
Parallel SSSP

- It is important to parallelize outer loop

- Some heuristics have been proposed:
  - All vertices with distances < threshold L can be removed
  - A large L can promote parallelism but can result in poor work efficiency due to unwanted computations due to reinsertions and repeated relaxations
Parallel Dijkstra’s SSSP

- Parallelism depends on graph topology
- Number of vertices that can be removed and processed in parallel
- Number of edges that can be relaxed in parallel
Bellman-Ford

- Larger parallelism, low work efficiency
- All edges are relaxed in all iterations till convergence

```
input: a graph graph(V, E) with N vertices in a set V, and M edges in a set, E, and M
weights, w of the edges. A source s
1 forall the v ∈ V do
2     dist[v] ←∞;
3 end
4 dist[s] ← 0;
5 for (i = 0; i < N; i++) do
6     forall the (u, v) edge ∈ E do
7         if (dist[u] + w(u, v)) < dist[v] then
8             dist[v] = dist[u] + w(u, v);
9         end
10 end
```

Fig. 4.22: Sequential Bellman-Ford SSSP Algorithm

- Suitable for GPUs; Inner loop needs to be protected by atomics
SSSP: Delta-stepping

- Balance between the two
- Balances work efficiency and parallelism
- Maintains tentative distances in buckets
- Each bucket maintains a range of tentative distances
- Range is given by delta
Delta stepping

- Assign source vertex to B0; all vertices to Binf
- Outer loop of phases; inner loop of steps
- In each phase, algorithms considers the non-empty bucket of lowest index
- Let Bj be such a bucket
- At the beginning of the phase, all vertices with final distances less than \((\text{delta}.j+1)\) would have been settled
Delta stepping

- Algorithm removes vertices from Bj and relaxes all its outgoing edges
- This can migrate vertices from higher indexed bucket to a lower index bucket, k, with k≥j
- Perform until Bj becomes empty
- What happens if delta=1, and delta=inf?
Delta stepping

---

**Input**: a graph $\text{graph}(V, E)$ with $N$ vertices in a set $V$, and $M$ edges in a set, $E$, and $M$ weights, $w$ of the edges. A source $s$

1. forall the $v \in V$ do
2.     $\text{dist}[v] \leftarrow \infty$ ;
3. end
4. $\text{dist}[s] \leftarrow 0$ ;
5. $B_0 = s$ ;
6. $B_\infty = v \in V, v \neq s$ ;
Delta stepping

```plaintext
while ∃ a non-empty bucket do
  /* For each phase*/
  Find the non-empty bucket, $B_j$, that is not empty;
  while $B_j ≠ ∅$ do
    /* For each step*/
    forall the $u ∈ B_j$ do
      /* Relax (u,v) edge*/
      forall the $v ∈ neighbors(u)$ do
        if $(\text{dist}[u] + \text{w}(u,v)) < \text{dist}[v]$ then
          oldDist = dist[v];
          $\text{dist}[v] = \text{dist}[u] + \text{w}(u,v);$;
          newDist = dist[v];
          $l = \frac{(\text{oldDist} - 1)}{\Delta};$
          /* old bucket index*/
          $k = \frac{(\text{newDist} - 1)}{\Delta};$
          /* new bucket index*/
          Migrate $v$ from $B_l$ to $B_k$;
      end
  end
end
```

Fig. 4.23: $\Delta$-stepping SSSP Algorithm
Delta-stepping Parallelization

- Inner loop is parallelized where all relaxations in a bucket are parallelized
- Shared memory parallelism – updates of the distances will have to be protected by atomics
Delta-stepping Parallelization

- Distributed memory parallelism
  - Vertices distributed
  - Non-empty buckets maintained by all processors
  - Each processor stores and processes only its vertices in its buckets
  - Allreduce for finding the lowest index bucket
  - Simultaneous relaxations using BSP (Bulk synchronous parallelism) model
SSSP on GPUs

- Most follow Bellman-Ford: Large number of edges processes by threads
- Two models:
  - Topology-driven: All vertices with non-infinite distances are processed by corresponding threads
  - Data-driven: Only those whose distances have changed in the previous iteration are processed. A work list is maintained.
Topology-driven Algorithm

```java
main()
{
  forall the v ∈ V do
  |
  dist[v] ← ∞ ;
  end
  dist[s] ← 0 ;
  change = 1 ;
  while change do
    change = 0 ;
    sssp_kernel(change) ;
  end
}

sssp_kernel(INOUT change){
  forall u ∈ V parallel
  |
  if u ≠ ∞ then
    for v ∈ neighbors(u) do
      if (dist[u] + w(u, v)) < dist[v] then
        dist[v] = dist[u] + w(u, v) ;
        change = 1 ;
      end
    end
  end
}

Fig. 4.24: Topology-driven SSSP Algorithm
Data-driven Algorithm

```
main()
  forall the v ∈ V do
  |dist[v] ← ← ∞ ;
  end
  dist[s] ← 0 ;
  Add s to worklist_in ;
  worklist_out = {} ;
  while worklist_in ≠ {} do
    sssp_kernel(worklist_in, worklist_out) ;
    copy worklist_out to worklist_in ;
    worklist_out = {} ;
  end

sssp_kernel(IN worklist_in, OUT worklist_out){
  for u ∈ worklist_in parallel
    for v ∈ neighbors(u) do
      if (dist[u] + w(u, v)) < dist[v] then
        atomicMin(dist[v], dist[u] + w(u, v)) ;
        Add v to worklist_out ;
      end
    end
  end
}
```

Fig. 4.25: Data-driven SSSP Algorithm
Pros and Cons

- Topology-driven: Low work efficiency
- Data-driven: Need atomics, atomicMin
- Can result in lost updates of minimum distances
Pros and Cons

- Topology-driven version does not need atomics
- This is because SSSP has *monotonicity* property: the distance value of a vertex is non-increasing
- Property utilized in topology-driven version to avoid atomics
Pros and Cons

- Since all active vertices with non-infinite distances are processed in all iterations, the lost updates will be reconsidered in the subsequent iterations.
- Even if there is a lost update, the thread with the minimum distance will get its chance in the next iteration.
Reducing Atomics in Atomic Addition to Worklist

- *Work chunking*: Perform atomic updates for a chunk of elements rather than for every element.
- Prefix sums can be used to avoid atomics.
- Prefix sums themselves can be hierarchically constructed: *hierarchical prefix sum*.
Redundancy in Worklists

- Two threads can add the same neighbour vertex to a worklist
- Duplicates can be huge!
- Two ways to avoid:
  - A post-processing *filtering* kernel
  - *Hash-based* culling; a thread uses hashing to find if its neighbour has been added
Load Balancing

- A major challenge among graph processing algorithms is the load imbalance among threads.
- Modern-day graphs in social networks are scale-free graphs.
- These graphs follow power-law distribution of degrees.

![Degree distribution of Twitter follower network [WWW 2010]
$|V| = 41.65M, |E| = 1468.36M$]
Load Balancing Strategies: Workload Decomposition

- Edges decomposed across threads in a block distribution
Load Balancing Strategies: Node Splitting

- Split each high degree node into multiple low-degree nodes called virtual nodes
Prim’s Minimal Spanning Tree, APSP using Dijkstra’s
(Book by Grama et al. – Pages starting from 432, then from 438)
Minimal Spanning Tree – Prim’s Algorithm

- Spanning tree of a graph, $G (V,E)$ - tree containing all vertices of $G$
- MST - spanning tree with minimum sum of weights
- Follows similar structure as Dijkstra’s SSSP.
- Vertices are added to a set $V_t$ that holds vertices of MST; Initially contains an arbitrary vertex, $r$, as root vertex.
Minimal Spanning Tree – Prim’s Algorithm

- An array \( d \) such that \( d[v \in (V-V_t)] \) holds weight of the edge with least weight between \( v \) and any vertex in \( V_t \); Initially \( d[v] = w[r,v] \)
- Find the vertex in \( d \) with minimum weight and add to \( V_t \)
- Update \( d \)
- Time complexity - \( O(n^2) \)
Parallelization

- Vertex V and d array partitioned across P processors
- Each processor finds local minimum in d
- Then global minimum across all d performed by reduction on a processor
- The processor finds the next vertex u, and broadcasts to all processors
Parallelization

- All processors update d; The owning processor of u marks u as belonging to $V_t$
- Process responsible for v must know $w[u,v]$ to update $d[v]$; 1-D block mapping of adjacency matrix
- Complexity - $O(n^2/P) + (On\log P)$ for communication
All-Pairs Shortest Paths

- To find shortest paths between all pairs of vertices
- Dijkstra’s algorithm for single-source shortest path can be used for all vertices
- Two approaches
All-Pairs Shortest Paths

- Source-partitioned formulation: Partition the vertices across processors
  - Works well if \( p \leq n \); No communication
  - Can at best use only \( n \) processors
  - Time complexity?

- Source-parallel formulation: Parallelize SSSP for a vertex across a subset of processors
  - Do for all vertices with different subsets of processors

- Hierarchical formulation
  - Exploits more parallelism
  - Time complexity?
Graph Partitioning
Graph Partitioning

- For many parallel graph algorithms, the graph has to be partitioned into multiple partitions and each processor takes care of a partition

- Criteria:
  - The partitions must be balanced (uniform computations)
  - The edge cuts between partitions must be minimal (minimizing communications)

- Some methods
  - BFS: Find BFS and descend down the tree until the cumulative number of nodes = desired partition size
  - Mostly: Multi-level partitioning based on coarsening and refinement (a bit advanced)
  - Another popular method: Kernighan-Lin
Partitioning without nodal coordinates - Kernighan/Lin

° Take a initial partition and iteratively improve it
  • Kernighan/Lin (1970), cost = $O(|N|^3)$ but easy to understand
  • Fiduccia/Mattheyses (1982), cost = $O(|E|)$, much better, but more complicated

° Let $G = (N, E, W_E)$ be partitioned as $N = A \cup B$, where $|A| = |B|$

° $T = \text{cost}(A,B) = \sum \{W(e) \text{ where } e \text{ connects nodes in } A \text{ and } B\}$

° Find subsets $X$ of $A$ and $Y$ of $B$ with $|X| = |Y|$ so that swapping $X$ and $Y$ decreases cost:
  - newA = $A - X \cup Y$ and newB = $B - Y \cup X$
  - newT = cost(newA, newB) < cost(A,B)
  - Keep choosing $X$ and $Y$ until cost no longer decreases

° Need to compute newT efficiently for many possible $X$ and $Y$, choose smallest
Kernighan/Lin - Preliminary Definitions

\( T = \text{cost}(A, B), \quad \text{new}T = \text{cost}(\text{new}A, \text{new}B) \)

- Need an efficient formula for \( \text{new}T \); will use
  - \( E(a) = \text{external cost of } a \text{ in } A = \sum \{ W(a,b) \text{ for } b \text{ in } B \} \)
  - \( l(a) = \text{internal cost of } a \text{ in } A = \sum \{ W(a,a') \text{ for other } a' \text{ in } A \} \)
  - \( D(a) = \text{cost of } a \text{ in } A = E(a) - l(a) \)
    - Moving \( a \) from \( A \) to \( B \) would decrease \( T \) by \( D(a) \)
  - \( E(b), l(b) \) and \( D(b) \) defined analogously for \( b \) in \( B \)

- Consider swapping \( X = \{a\} \) and \( Y = \{b\} \)
  - \( \text{new}A = A - \{a\} \cup \{b\}, \quad \text{new}B = B - \{b\} \cup \{a\} \)

- \( \text{new}T = T - (D(a) + D(b) - 2*W(a,b)) = T - \text{gain}(a,b) \)
  - \( \text{gain}(a,b) \) measures improvement gotten by swapping \( a \) and \( b \)

- Update formulas, after \( a \) and \( b \) are swapped
  - \( \text{new}D(a') = D(a') + 2*W(a',a) - 2*W(a',b) \) for \( a' \) in \( A \), \( a' \neq a \)
  - \( \text{new}D(b') = D(b') + 2*W(b',b) - 2*W(b',a) \) for \( b' \) in \( B \), \( b' \neq b \)
Kernighan/Lin Algorithm

Compute $T = \text{cost}(A, B)$ for initial $A, B$  
... cost = $O(|N|^2)$

Repeat

... One pass greedily computes $|N|/2$ possible $X, Y$ to swap, picks best

Compute costs $D(n)$ for all $n$ in $N$  
... cost = $O(|N|^2)$

Unmark all nodes in $N$  
... cost = $O(|N|)$

While there are unmarked nodes

  Find an unmarked pair $(a, b)$ maximizing gain$(a, b)$  
  ... cost = $O(|N|^2)$

  Mark $a$ and $b$ (but do not swap them)  
  ... cost = $O(1)$

  Update $D(n)$ for all unmarked $n$, 
  as though $a$ and $b$ had been swapped  
  ... cost = $O(|N|)$

Endwhile

... At this point we have computed a sequence of pairs

... $(a_1, b_1), \ldots, (a_k, b_k)$ and gains gain$(1), \ldots, \text{gain}(k)$

... where $k = |N|/2$, numbered in the order in which we marked them

Pick $m$ maximizing $\text{Gain} = \sum_{k=1}^{m} \text{gain}(k)$  
... cost = $O(|N|)$

... Gain is reduction in cost from swapping $(a_1, b_1)$ through $(a_m, b_m)$

If Gain > 0 then  
... it is worth swapping

  Update new$A = A - \{ a_1, \ldots, a_m \} \cup \{ b_1, \ldots, b_m \}$  
  ... cost = $O(|N|)$

  Update new$B = B - \{ b_1, \ldots, b_m \} \cup \{ a_1, \ldots, a_m \}$  
  ... cost = $O(|N|)$

  Update $T = T - \text{Gain}$  
  ... cost = $O(1)$

endif

Until Gain <= 0
Parallel partitioning

- Can use divide and conquer strategy
- A master node creates two partitions
- Keeps one for itself and gives the other partition to another processor
- Further partitioning by the two processors and so on...
Multi-level partitioning
K-way multilevel partitioning algorithm

- Has 3 phases: coarsening, partitioning, refinement (uncoarsening)
- Coarsening - a sequence of smaller graphs constructed out of an input graph by collapsing vertices together
Coarsening

- Formulated as a maximal matching problem
- Matching – finding a set of non-adjacent edges, i.e., edges are not incident on same vertices
- Maximal matching: A matching where addition of one more edge results in the loss of matching property
- Commonly used heuristic: heaviest edge matching
K-way multilevel partitioning algorithm

- When enough vertices are collapsed together so that the coarsest graph is sufficiently small, a k-way partition is found.

- Finally, the partition of the coarsest graph is projected back to the original graph by refining it at each uncoarsening level using a k-way partitioning refinement algorithm.
K-way partitioning refinement

- A simple randomized algorithm that moves vertices among the partitions to minimize edge-cut and improve balance
- For a vertex v, let neighborhood N(v) be the union of the partitions to which the vertices adjacent to v belong
- In a k-way refinement algorithm, vertices are visited randomly
K-way partitioning refinement

- A vertex v is moved to one of the neighboring partitions N(v) if any of the following vertex migration criteria is satisfied
  - The edge-cut is reduced while maintaining the balance
  - The balance improves while maintaining the edge-cut

- This process is repeated until no further reduction in edge-cut is obtained
Graph Coloring
Graph Coloring Problem

- Given $G(A) = (V, E)$
- $\sigma: V \rightarrow \{1, 2, ..., s\}$ is an $s$-coloring of $G$ if $\sigma(i) \neq \sigma(j)$ for every $(i, j)$ edge in $E$
- Minimum possible value of $s$ is the chromatic number of $G$
- Graph coloring problem is to color nodes with the chromatic number of colors
- NP-complete problem
Parallel graph Coloring – General algorithm

\[
\begin{align*}
\text{ParallelColoring}(G = (V, E)) \quad \text{begin} \\
\quad U & \leftarrow V \\
\quad G'' & \leftarrow G' \\
\quad \text{while } (G'' \text{ is not empty}) \text{ do in parallel} \\
\quad \quad \text{Find an independent set } I \text{ in } G'' \\
\quad \quad \text{Color the vertices in } I \\
\quad \quad U & \leftarrow U \setminus I \\
\quad \quad G'' & \leftarrow \text{graph induced by } U \\
\quad \text{end-while} \\
\text{end}
\end{align*}
\]
Parallel Graph Coloring – Finding Maximal Independent Sets – Luby (1986)

I = null
V' = V
G' = G

While G' ≠ empty
Choose an independent set I' in G'
I = I U I';  X = I' U N(I')  (N(I') – adjacent vertices to I')
V' = V' \ X;  G' = G(V')
end

For choosing independent set I’: (Monte Carlo Heuristic)
1. For each vertex, v in V' determine a distinct random number p(v)
2. v in I iff p(v) > p(w) for every w in adj(v)

Color each MIS a different color

Disadvantage:
- Each new choice of random numbers requires a global synchronization of the processors.
Parallel Graph Coloring – Gebremedhin and Manne (2003)

```
BlockPartitionBasedColoring(G, p)
begin
  1. Partition V into p equal blocks V_1 ... V_p, where \( \lfloor \frac{n}{p} \rfloor \leq |V_i| \leq \lceil \frac{n}{p} \rceil \)
     for i = 1 to p do in parallel
       for each v_j \in V_i do
         assign the smallest legal color to vertex v_j
         barrier synchronize
       end-for
     end-for
end-for
```
Sources/References

Community Detection

- Given a graph, the goal is to partition into communities such that related vertices are assigned to the same community.
Metric

- Modularity – Measure to evaluate the goodness of a community
  - Measures the fraction of edges that lie within the community
  - Measures the difference between fraction of edges within communities compared to the expected fraction that would exist on a random graph with identical vertex and degree distributions

\[
Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i * k_j}{2m}) \delta(c_i, c_j)
\]

where:

- \( m \) = sum of all the edge-weights
- \( k_i \) = weighted degree of vertex \( i \)
- \( c_i \) = community that contains vertex \( i \)
- \( \delta(c_i, c_j) = 1 \) if \( c_i = c_j \), 0 otherwise.
Modularity

\[ Q = \sum_{c \in C} \left[ \frac{e_{ij}}{2m} - \left( \frac{a_c}{2m} \right)^2 \right] \]

where:

\[ e_{ij} = \sum w_{ij} : \forall i, j \in c, \text{ and } \{i, j\} \in E \]

\[ a_c = \sum_{i \in c} k_i \]
Louvain Method

- Multi-phase, multi-iteration heuristic
- Iteratively improves the quality of the community until the gain in quality becomes negligible
- Complete sweep of a graph per iteration
- Graph coarsenings between phases
Louvain Method

- Each phase runs for a number of iterations until convergence
- Initially, each vertex is a community
- In each iteration:
  - Gain in modularity calculated when moving a vertex to each of its neighboring communities
    - If positive gain moved
- Iterations continued until convergence
- At the end of the phase, the vertices are collapsed
Sequential Algorithm

Algorithm 1: Serial Louvain algorithm.
Input: Graph $G = (V, E)$, threshold $\tau$
Input: Initial community assignment, $C_{init}$

1: $Q_{prev} \leftarrow -\infty$
2: $C_{prev} \leftarrow$ Initialize each vertex in its own community
3: while true do
4:     for all $v \in V$ do
5:         $N(v) \leftarrow$ neighboring communities of $v$
6:         $targetComm \leftarrow \text{arg max}_{t \in N_v} \Delta Q(v \text{ moving to } t)$
7:         if the gain is positive then
8:             Move $v$ to $targetComm$ and update $C_{curr}$
9:         $Q_{curr} \leftarrow \text{ComputeModularity}(V, E, C_{curr})$
10:        if $Q_{curr} - Q_{prev} \leq \tau$ then
11:            break
12:        else
13:            $Q_{prev} \leftarrow Q_{curr}$
Challenges in Parallel Algorithm

- Lag of Community updates
- Significant communication overhead at every iteration of every phase
- Modularity calculation requires global accumulation of weights, hence global collectives
- New vertex-community mapping must be communicated at the end of every phase
Parallel Louvain Algorithm

1: \textbf{function} LOUVAIN\textsc{iteration}(G_i, C_{\text{curr}}) \\
2: \hspace{1em} V_g \leftarrow \text{ExchangeGhostVertices}(G_i) \\
3: \hspace{1em} \textbf{while} \text{true} \textbf{do} \\
4: \hspace{2em} \text{send} \hspace{1em} \text{latest} \hspace{1em} \text{information} \hspace{1em} \text{on} \hspace{1em} \text{those} \hspace{1em} \text{local} \hspace{1em} \text{vertices} \hspace{1em} \text{that} \hspace{1em} \text{are} \hspace{1em} \text{stored} \hspace{1em} \text{as} \hspace{1em} \text{ghost} \hspace{1em} \text{vertices} \hspace{1em} \text{on} \hspace{1em} \text{remote} \hspace{1em} \text{processes} \\
5: \hspace{2em} \text{receive} \hspace{1em} \text{latest} \hspace{1em} \text{information} \hspace{1em} \text{on} \hspace{1em} \text{all} \hspace{1em} \text{ghost} \hspace{1em} \text{vertices} \\
6: \hspace{2em} \textbf{for} \hspace{1em} v \in V_i \hspace{1em} \textbf{do} \\
7: \hspace{3em} \text{Compute} \Delta Q \text{ that can be achieved by moving} \hspace{1em} v \hspace{1em} \text{to each} \hspace{1em} \text{of its neighboring} \hspace{1em} \text{communities} \\
8: \hspace{3em} \text{Determine} \text{target} \hspace{1em} \text{community} \hspace{1em} \text{for} \hspace{1em} v \hspace{1em} \text{based} \hspace{1em} \text{on} \hspace{1em} \text{the} \hspace{1em} \text{migration} \hspace{1em} \text{that} \hspace{1em} \text{maximizes} \Delta Q \\
9: \hspace{3em} \text{Update} \hspace{1em} \text{community} \hspace{1em} \text{information} \hspace{1em} \text{for both the source} \hspace{1em} \text{and} \hspace{1em} \text{destination} \hspace{1em} \text{communities} \hspace{1em} \text{of} \hspace{1em} v \\
10: \hspace{2em} \text{send} \hspace{1em} \text{updated} \hspace{1em} \text{information} \hspace{1em} \text{on} \hspace{1em} \text{ghost} \hspace{1em} \text{communities} \hspace{1em} \text{to} \hspace{1em} \text{owner} \hspace{1em} \text{processes} \\
11: \hspace{2em} C_{\text{info}} \leftarrow \text{receive} \hspace{1em} \text{and} \hspace{1em} \text{update} \hspace{1em} \text{information} \hspace{1em} \text{on} \hspace{1em} \text{local} \hspace{1em} \text{communities} \\
12: \hspace{1em} \text{currMod}_i \leftarrow \text{Compute} \hspace{1em} \text{modularity} \hspace{1em} \text{based} \hspace{1em} \text{on} \hspace{1em} G_i \hspace{1em} \text{and} \hspace{1em} C_{\text{info}} \\
13: \hspace{1em} \text{currMod} \leftarrow \text{all-reduce: } \sum_{V_i} \text{currMod}_i \\
14: \hspace{1em} \text{if} \hspace{1em} \text{currMod} - \text{prevMod} \leq \tau \hspace{1em} \text{then} \\
15: \hspace{2em} \text{break} \\
16: \hspace{1em} \text{prevMod} \leftarrow \text{currMod} \\
17: \hspace{1em} \text{return} \hspace{1em} \text{prevMod}
Optimizations

- One of the major contributors of communication is the communication of ghost vertex information.

- Observation: Rate of modularity increase decreases with the number of iterations – diminishing benefits.

- This fact can be used to drop out certain vertices from computations and communications.

- Mark vertices as active and inactive probabilistically.
Optimizations

- If the vertex has not moved recently, the probability that it will stay active is reduced

- e.g.:

\[
P_{v,k} = \begin{cases} 
P_{v,k-1} \times (1 - \alpha), & \text{if } C_{v,k-1} = C_{v,k-2} \\
1, & \text{otherwise}
\end{cases}
\]  
(3)
Optimizations within a node

- Within a node, concurrent updates need locking
- Can identify non-colliding vertices and update them concurrently without locks?
- How?
Paper: Distributed Louvain Algorithm for Graph Community Detection