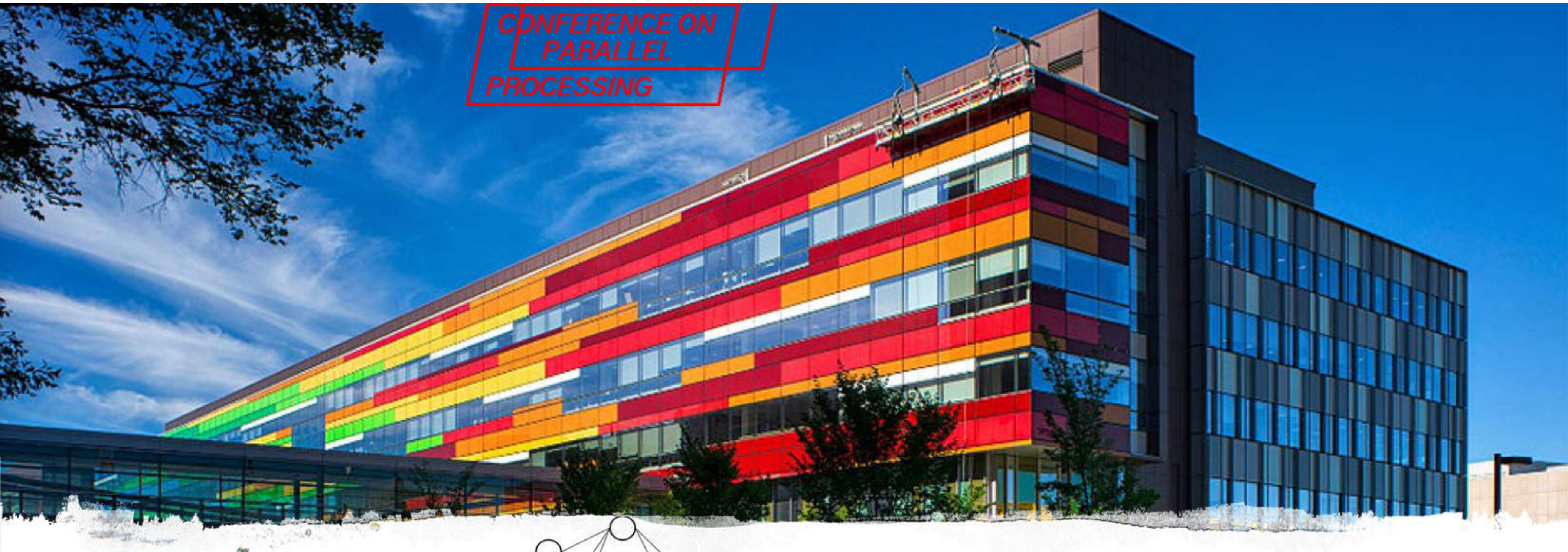
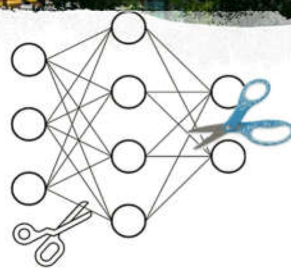




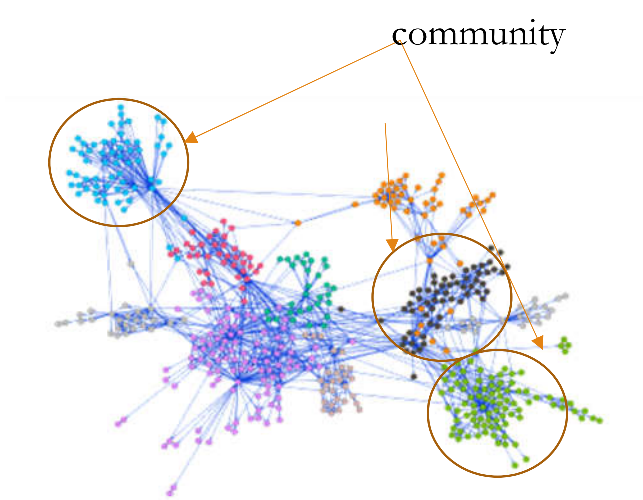
Prune the Unnecessary: Parallel Pull-Push Louvain Algorithms with Automatic Edge Pruning

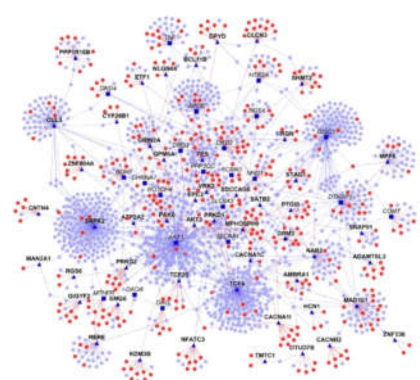
Jesmin Jahan Tithi ♥
Andrzej Stasiak *
Sriram Aananthakrishnan *
Fabrizio Petrini ♥
♥Parallel Computing Labs, Intel,
*Data Center Group, Intel.

What is community?

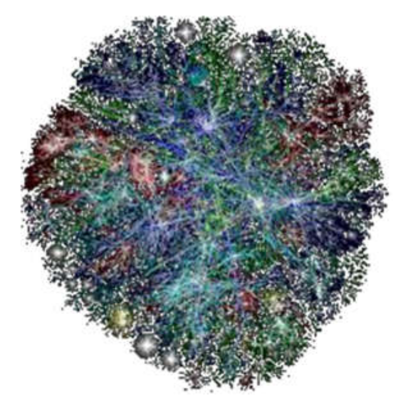
What is Community?

- Sets of vertices that have dense intra-connections, but sparse inter-connections
- Uncover hidden structures inside a graph in a form of coherent modules of vertices
- Strongly correlated to functional and structural properties





Protein-Protein Interaction Network



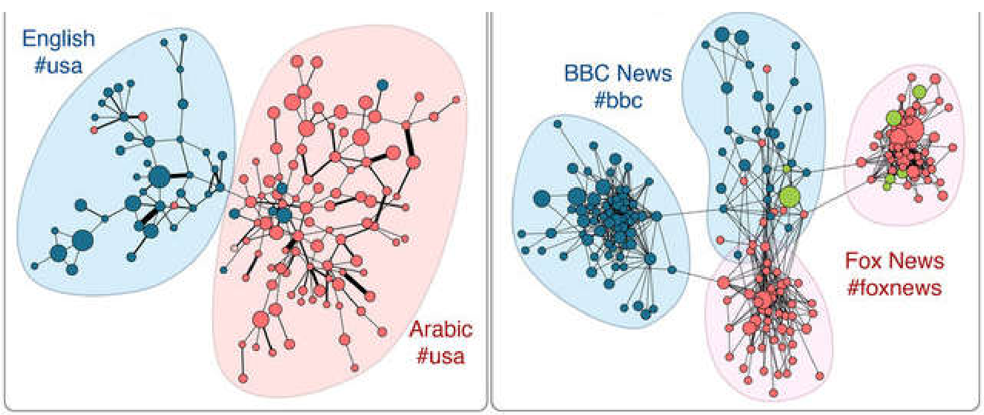
World Wide Web

Image source: Google Image

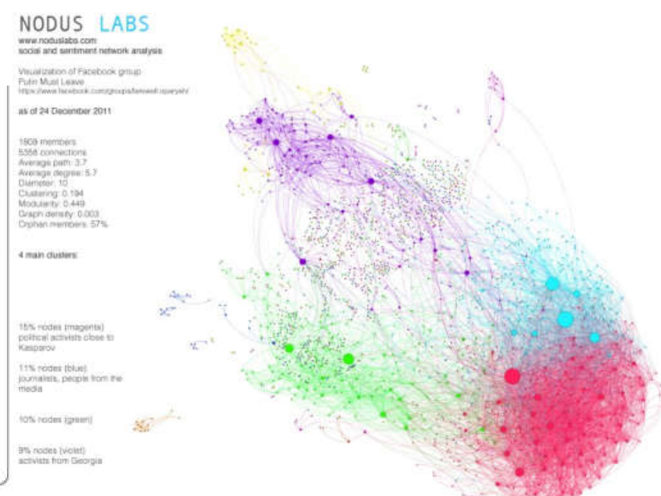
What is community detection?

What is Community Detection?

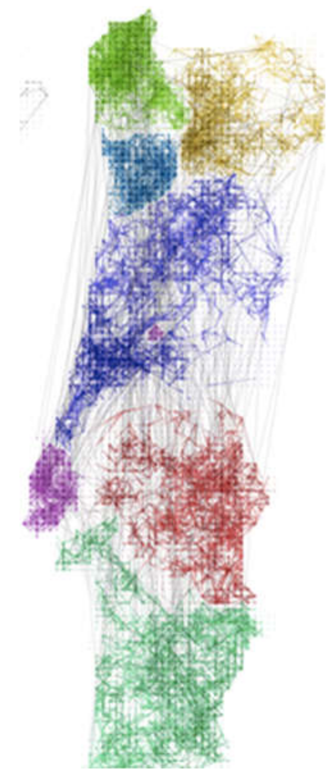
- Algorithms to identify communities in a network
- **Applications:** network analysis to retrieve information or patterns of the network



Virality Prediction and Community Structure in Social Networks



Nodus Labs Against Putin Facebook protest group visualization, December 2011



http://senseable.mit.edu/community_detection/

How to measure the quality of the detected communities?

A Measure of Solution Quality

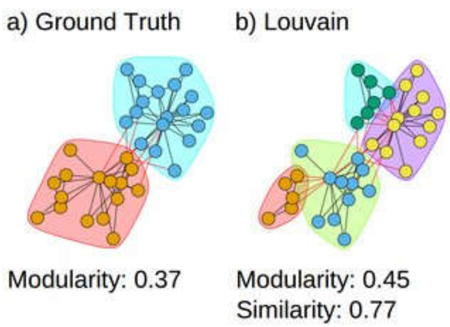
- **Modularity:** A measure of interconnectedness of the communities

$$Modularity, Q = \sum_{c \in C} \left[\frac{\sum c_{in}}{2m} - \frac{\sum {c_{tot}}^2}{4m^2} \right] \text{Max Value of Q } = 1$$

$$\sum c_{in} = \sum W_{u,v}, for\ all\ u, v \in c$$

$$\sum c_{tot} = \sum W_{u,v}, for\ all\ u \in c\ or\ v \in c$$

$$m = \sum_{e(u,v)} W_{u,v}$$



- $|Q| \in (0, 1]$, and the higher the better
- Community detection algorithm identifies communities in a way that maximizes modularity

How do we maximize modularity?

A Recipe of Modularity Optimization

- **Modularity:** A measure of interconnectedness of the communities

$$Modularity, Q = \sum_{c \in C} \left[\frac{\sum c_{in}}{2m} - \frac{\sum {c_{tot}}^2}{4m^2} \right] \text{ Max Value of Q } = 1$$

- Large values of Q correlate with **high $\sum c_{in}$** and low $\sum c_{tot}$
  - Communities that are dense within their structure and weakly coupled among each other

- To **get high $\sum c_{in}$**, the highest possible number of edges should fall in each community

A Recipe of Modularity Optimization

- **Modularity:** A measure of interconnectedness of the communities

$$Modularity, Q = \sum_{c \in C} \left[\frac{\sum c_{in}}{2m} - \frac{\sum c_{tot}^2}{4m^2} \right]$$ Max Value of Q $= 1$

- Large values of Q correlate with high $\sum c_{in}$ and **low $\sum c_{tot}$**
    - Communities that are dense within their structure and weakly coupled among each other

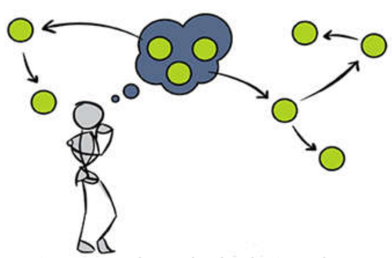
- To **decrease $\sum c_{tot}$**, divide the network into several communities with small total degrees

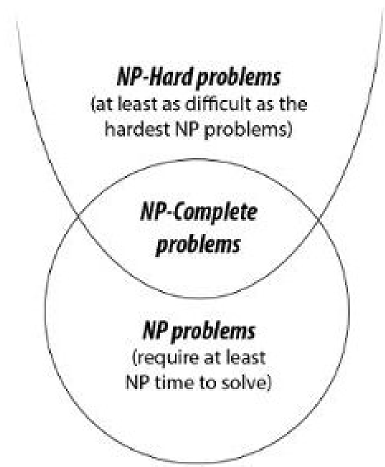
NP-hardness of Modularity Optimization

- **Modularity:** A measure of interconnectedness of the communities

$$Modularity, Q = \sum_{c \in C}\left[\frac{\sum c_{in}}{2m} - \frac{\sum c_{tot}^{2}}{4m^{2}}\right] \quad \text{Max Value of Q} = 1$$

Challenge: Finding communities with optimal modularity is "NP-hard"





Louvain

Maximizes modularity following a greedy algorithm

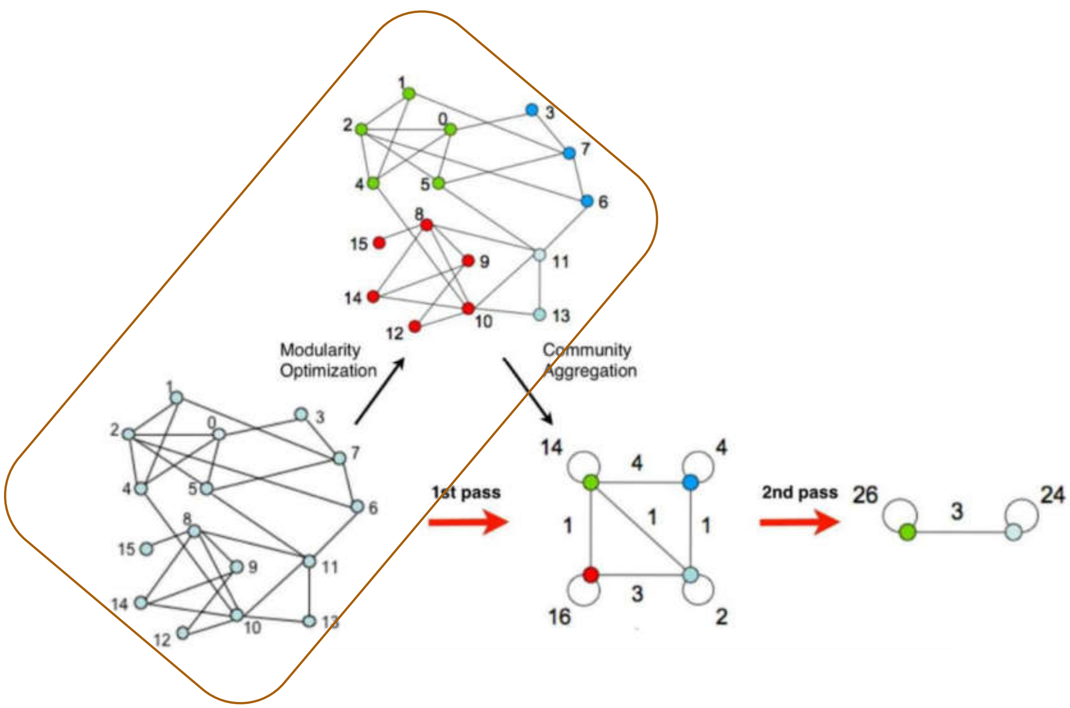
V. D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech. (2008) P10008,* p. 12, 2008

Louvain: Algorithm Steps

- **Outer Loop:** Traverse the graph in several passes to incrementally build communities

Louvain: Algorithm Steps

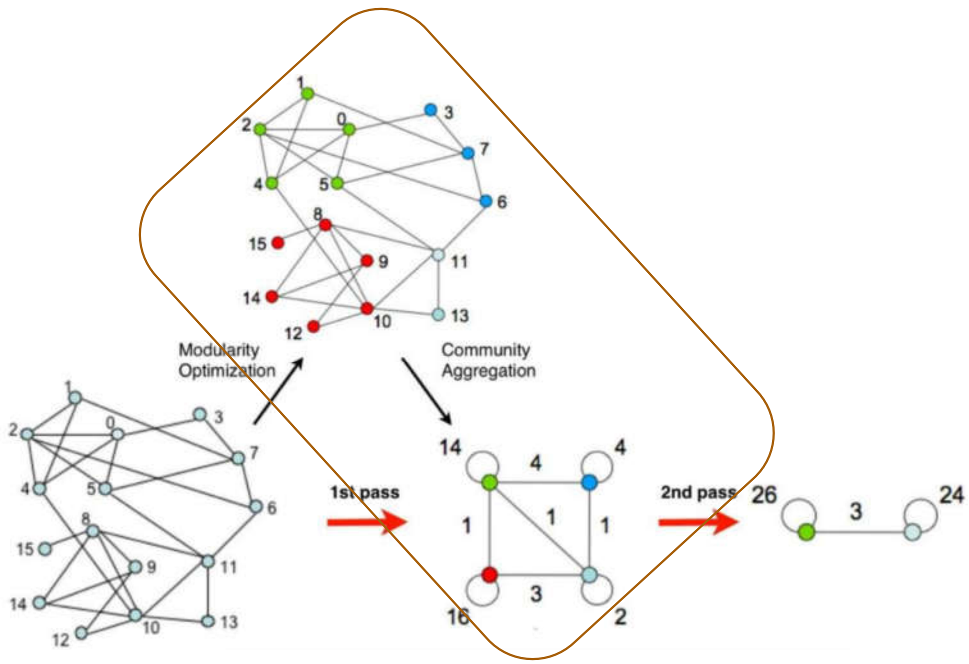
- **Outer Loop:** Traverse the graph in several passes to incrementally build communities
  - **Phase 1:** Modularity Optimization/Inner loop



V. D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech. (2008) P10008,* p. 12, 2008

Louvain: Algorithm Steps

- **Outer Loop:** Traverse the graph in several passes to incrementally build communities
  - **Phase 2:** Community Aggregation and Graph Reconstruction



V. D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech. (2008) P10008,* p. 12, 2008

Louvain: Algorithm Steps

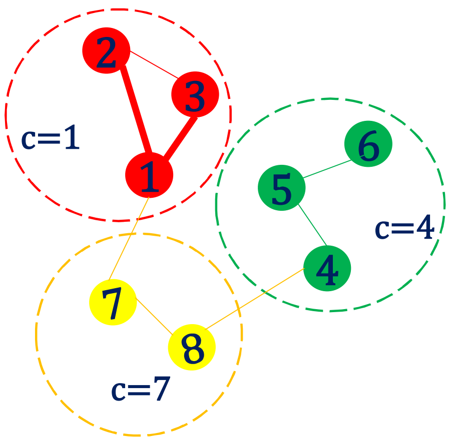
- **Outer Loop:** Traverse the graph in several passes to incrementally build communities
  - **Phase 1:** Modularity Optimization/Inner loop - $O\big(k(|V| + |E|)\big)$

  - **Phase 2:** Community Aggregation and Graph Reconstruction - $O(|V| + |E|)$

A key data structure to decide pull or push

Hash map NCW– ⟨ community_id, Some of edge weights ⟩

A hash map with ⟨key = neighboring community, val = sum of edge weights to that community⟩



Vertex 1 is neighbor to 2, 3 (members of community 1) => sum of edges weights=2

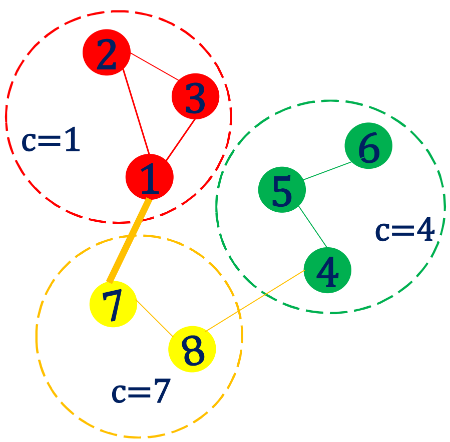
Vertex 1 is neighbor to 7 (member of community 7) => sum of edge weight = 1

⟨ community_id, Some of edge weights ⟩

$$NCW_1 = [\langle c=1, w_{1\rightarrow 1}=2 \rangle, \langle c=7, w_{1\rightarrow 7}=1 \rangle]$$

Hash map NCW– ⟨ community_id, Some of edge weights ⟩

A hash map with ⟨key = neighboring community, val = sum of edge weights to that community⟩



Vertex 1 is neighbor to 2, 3 (members of community 1) => sum of edges weights=2

Vertex 1 is neighbor to 7 (member of community 7) => sum of edge weight = 1

⟨ community_id, Some of edge weights ⟩

NCW_1=[⟨ c=1, $w_{1\rightarrow 1}$=2 ⟩, ⟨c=7, $w_{1\rightarrow 7}$=1⟩]

Louvain Pseudocode

Repeat if there is a change in community membership

Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while *true* **do**
        Qprev = Q
        for *all* $u \in V$ **do**
            //build NCW
            $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \; c \in NC_u$
            (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
            if *bestGain>0 and bestcomm* $\neq$ *oldcomm of u* **then**
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if $Q - Qprev \leq \tau$ **then**
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if *changes = 0* **then**
        break
    reconstructGraph(V, E, C)

Louvain Pseudocode

Initialize each vertex in its own community
Compute initial modularity

Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
  changes=0
  C = initialize each vertex in its own community
  Q = computeModularity(V, E, C)
  //Phase 1 (Inner loop): Modularity Optimization
  while *true* **do**
    Qprev = Q
    for $all\ u \in V$ **do**
      //build NCW
      $NCW_u = \langle c, w_{u \to c} \rangle\ \forall\ c \in NC_u$
      (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
      if *bestGain>0 and bestcomm ≠ oldcomm of u* **then**
        Move u to bestcomm and update C
    // new modularity value
    Q = computeModularity(V, E, C)
    if $Q - Qprev \leq \tau$ **then**
      break
    changes=1
  //Phase 2: Aggregation and Graph Reconstruction
  if *changes = 0* **then**
    break
  reconstructGraph(V, E, C)

Louvain Pseudocode

Phase 1/ inner loop starts

Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while *true* **do**
        Qprev = Q
        for *all* $u \in V$ **do**
            //build NCW
            $NCW_u = \langle c, w_{u \to c} \rangle \ \forall \ c \in NC_u$
            (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
            if *bestGain>0 and bestcomm ≠ oldcomm of u* **then**
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if $Q - Qprev \le \tau$ **then**
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if *changes = 0* **then**
        break
    reconstructGraph(V, E, C)

Louvain Pseudocode

For each vertex, build NCW by pulling community info from neighbors

Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

```
//Outer Loop
while true do
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while true do
        Qprev = Q
        for all u ∈ V do
            //build NCW
            NCW_u = ⟨c, w_{u→c}⟩ ∀ c ∈ NC_u
            (bestcomm, bestGain) =
              arg max_{c∈NCW_u} ΔQ_{u→c}
            if bestGain>0 and bestcomm ≠ oldcomm of u
              then
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if Q − Qprev ≤ τ then
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if changes = 0 then
        break
    reconstructGraph(V, E, C)
```

Louvain Pseudocode

Find the best community to move into by iterating though all entries of NCW

Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.

Result: Final Community Assignment C, and Modularity Q

```
//Outer Loop
while true do
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while true do
        Qprev = Q
        for all u ∈ V do
            //build NCW
            NCW_u = ⟨c, w_{u→c}⟩ ∀ c ∈ NC_u
            (bestcomm, bestGain) =
             arg max_{c∈NCW_u} ΔQ_{u→c}
            if bestGain>0 and bestcomm ≠ oldcomm of u
             then
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if Q − Qprev ≤ τ then
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if changes = 0 then
        break
    reconstructGraph(V, E, C)
```

Louvain Pseudocode

Move to the best community and update community info

Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.

Result: Final Community Assignment C, and Modularity Q

```
//Outer Loop
while true do
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while true do
        Qprev = Q
        for all u ∈ V do
            //build NCW
            NCW_u = ⟨c, w_{u→c}⟩ ∀ c ∈ NC_u
            (bestcomm, bestGain) =
             arg max_{c∈NCW_u} ΔQ_{u→c}
            if bestGain>0 and bestcomm ≠ oldcomm of u
             then
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if Q − Qprev ≤ τ then
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if changes = 0 then
        break
    reconstructGraph(V, E, C)
```

Louvain Pseudocode

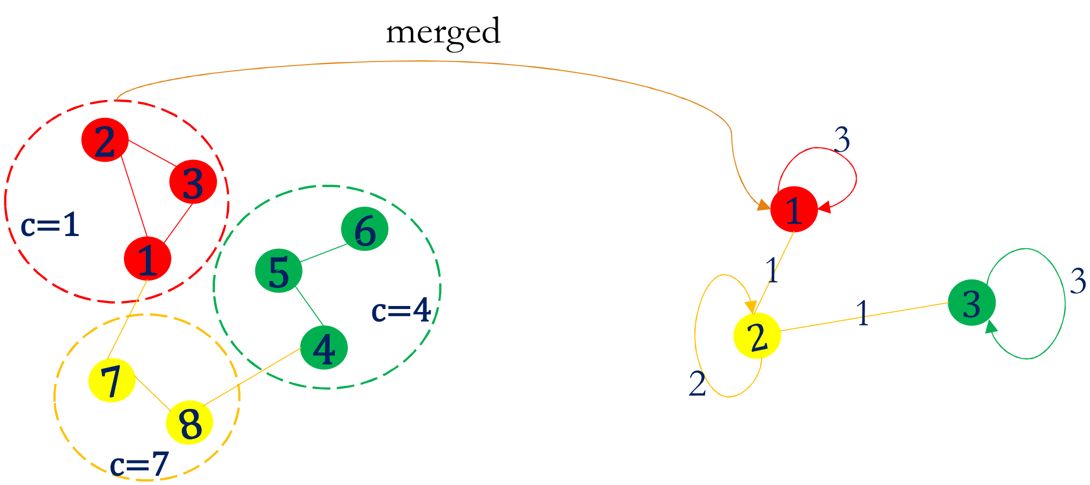
Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

```
//Outer Loop
while true do
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while true do
        Qprev = Q
        for all u ∈ V do
            //build NCW
            NCW_u = ⟨c, w_{u→c}⟩ ∀ c ∈ NC_u
            (bestcomm, bestGain) =
              arg max_{c∈NCW_u} ΔQ_{u→c}
            if bestGain>0 and bestcomm ≠ oldcomm of u
              then
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if Q − Qprev ≤ τ then
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if changes = 0 then
        break
    reconstructGraph(V, E, C)
```

Once done for all vertices, compute new modularity and repeat if modularity increased by a threshold

Louvain Pseudocode

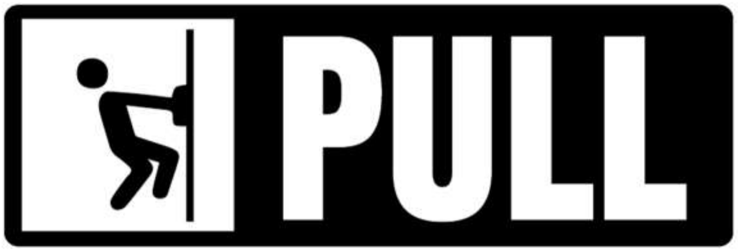


Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while *true* **do**
        Qprev = Q
        for *all* $u \in V$ **do**
            //build NCW
            $NCW_u = \langle c, w_{u \to c} \rangle \ \forall \ c \in NC_u$
            (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
            if *bestGain>0 and bestcomm ≠ oldcomm of u* **then**
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if $Q - Qprev \leq \tau$ **then**
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if *changes = 0* **then**
        break
    reconstructGraph(V, E, C)

When modularity stabilizes, create a new graph by merging all vertices in same community into one



We call the standard Louvain Algorithm a Pull-based Louvain Algorithm

To build NCW at each iteration, it pulls latest info from neighbors

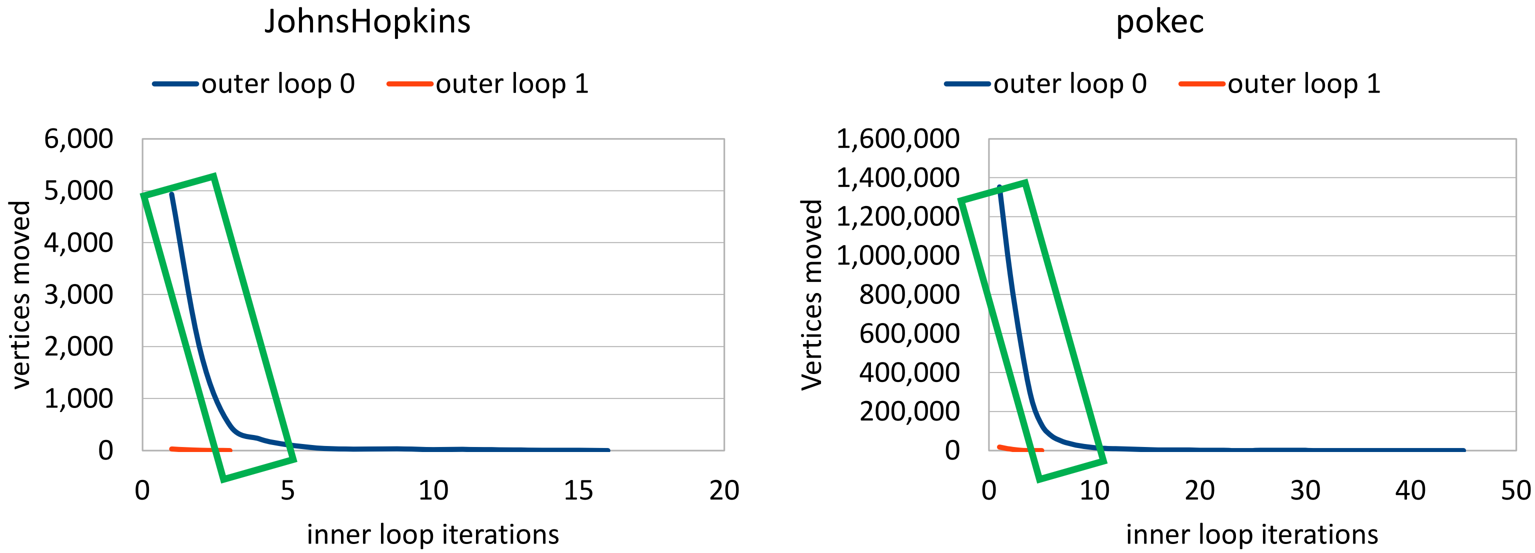
Algorithm 1: Sequential Louvain Algorithm.

```
Data: Graph G = (V, E), τ = Threshold.
Result: Final Community Assignment C, and Modularity Q
//Outer Loop
while true do
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while true do
        Qprev = Q
        for all u ∈ V do
            //build NCW
            NCW_u = ⟨c, w_{u→c}⟩ ∀ c ∈ NC_u
            (bestcomm, bestGain) =
              arg max_{c∈NCW_u} ΔQ_{u→c}
            if bestGain>0 and bestcomm ≠ oldcomm of u
              then
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if Q − Qprev ≤ τ then
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if changes = 0 then
        break
    reconstructGraph(V, E, C)
```

Unnecessary work in Louvain

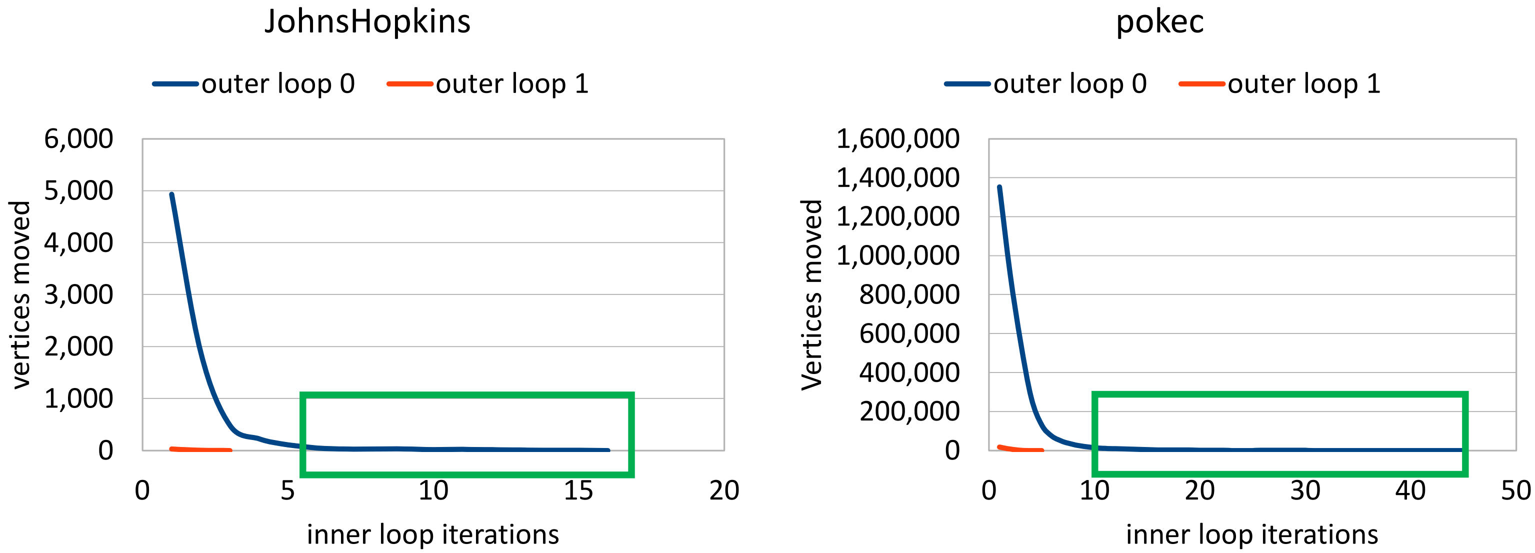
Observations

Number of vertex moves drops significantly after the first few iterations of phase1



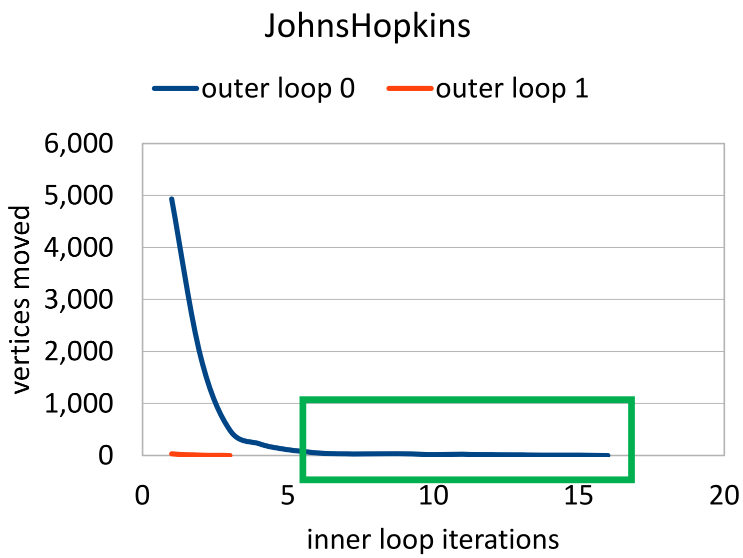
- For a particular outer loop, the number of vertices that change communities drops drastically after the first few inner loop iterations (e.g., 5).

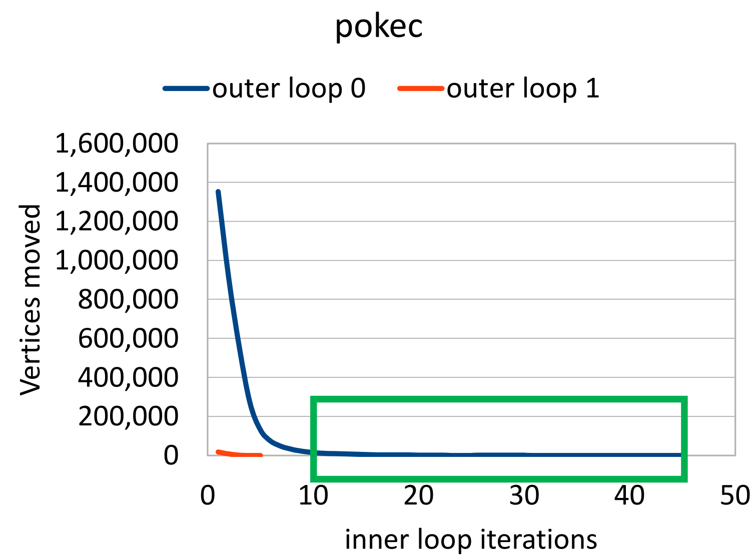
Number of vertex moves drops significantly after the first few iterations of phase1



- The number of vertices that change communities in the later inner loop iterations is minimal

Implications





- Wasteful to scan all neighbors to compute NCW, if no change in neighborhood
- Wasteful to iterate over all vertices for each iteration of phase 1, vertices do not move

Pruning Unnecessary Work in Louvain

Prune vertices that are unlikely to move

Prune unnecessary neighborhood exploration

Push-based Louvain Algorithm

Vertex does not pull, rather neighbors actively push any changes

Push-based Louvain

The Push-based algorithm starts with an initialized NCW, assuming each vertex is in its own community

Algorithm 2: A push-based sequential Louvain algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
  changes=0 C = initialize each vertex in its own community
  Q = computeModularity(V, E, C)
  //build initial NCW_u for all vertices
  $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \, c \in NC_u \; \forall \, c \in V$
  //Phase 1 (Inner loop): Modularity Optimization;
  while *true* **do**
    Qprev = Q
    for *all* $u \in V$ **do**
      (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
      if *bestGain>0 and bestcomm ≠ oldcomm of u* **then**
        Move u to bestcomm and update C
        Update NCW_u for bestcomm and oldcomm
        Update $NCW_x \; \forall \, x \in N_u$, for bestcomm and oldcomm
    // new modularity value
    Q = computeModularity(V, E, C)
    if $Q - Qprev \leq \tau$ **then**
      break
    changes=1
  //Phase 2: Aggregation and Graph Reconstruction
  if *changes = 0* **then**
    break
  reconstructGraph(V, E, C)

Push-based Louvain

During Phase 1, it never recreates NCW

Algorithm 2: A push-based sequential Louvain algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
    changes=0 C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //build initial NCW_u for all vertices
    $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \; c \in NC_u \; \forall \; c \in V$
    //Phase 1 (Inner loop): Modularity Optimization;
    while *true* **do**
        Qprev = Q
        for *all* $u \in V$ **do**
            (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
            if *bestGain>0 and bestcomm* $\neq$ *oldcomm of u* **then**
                Move u to bestcomm and update C
                Update NCW_u for bestcomm and oldcomm
                Update $NCW_x \; \forall \; x \in N_u$, for bestcomm and oldcomm
        // new modularity value
        Q = computeModularity(V, E, C)
        if $Q - Qprev \leq \tau$ **then**
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if *changes = 0* **then**
        break
    reconstructGraph(V, E, C)

Push-based Louvain

If there is a change in community membership

Algorithm 2: A push-based sequential Louvain algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
    changes=0 C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //build initial NCW_u for all vertices
    $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \; c \in NC_u \; \forall \; c \in V$
    //Phase 1 (Inner loop): Modularity Optimization;
    while *true* **do**
        Qprev = Q
        for *all* $u \in V$ **do**
            (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
            if *bestGain>0 and bestcomm* $\neq$ *oldcomm of u* **then**
                Move u to bestcomm and update C
                Update NCW_u for bestcomm and oldcomm
                Update $NCW_x \; \forall \; x \in N_u$, for bestcomm and oldcomm
        // new modularity value
        Q = computeModularity(V, E, C)
        if $Q - Qprev \leq \tau$ **then**
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if *changes = 0* **then**
        break
    reconstructGraph(V, E, C)

Push-based Louvain

Update NCW for the vertex itself, and push updates to all its neighbors

Algorithm 2: A push-based sequential Louvain algorithm.

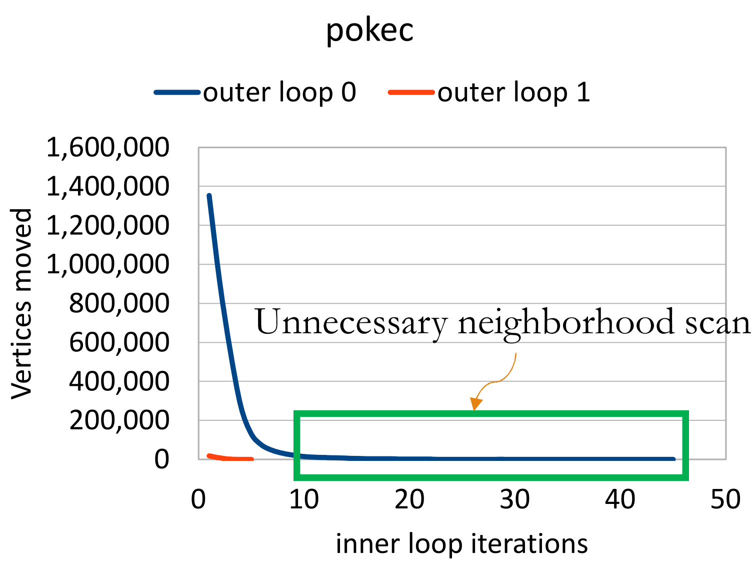
Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
    changes=0 C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //build initial NCW_u for all vertices
    $NCW_u = \langle c, w_{u \to c} \rangle \ \forall \ c \in NC_u \ \forall \ c \in V$
    //Phase 1 (Inner loop): Modularity Optimization;
    while *true* **do**
        Qprev = Q
        for *all* $u \in V$ **do**
            (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
            if *bestGain>0 and bestcomm* $\neq$ *oldcomm of u* **then**
                Move u to bestcomm and update C
                Update NCW_u for bestcomm and oldcomm
                Update $NCW_x \ \forall \ x \in N_u$, for bestcomm and oldcomm
        // new modularity value
        Q = computeModularity(V, E, C)
        if $Q - Qprev \leq \tau$ **then**
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if *changes = 0* **then**
        break
    reconstructGraph(V, E, C)

Pros and Cons of Pull and Push

Pull – Cons

Does redundant memory read
by scanning all vertices and their neighbors to rebuild
NCW for each inner loop,
even when the vertex's neighborhood has not changed



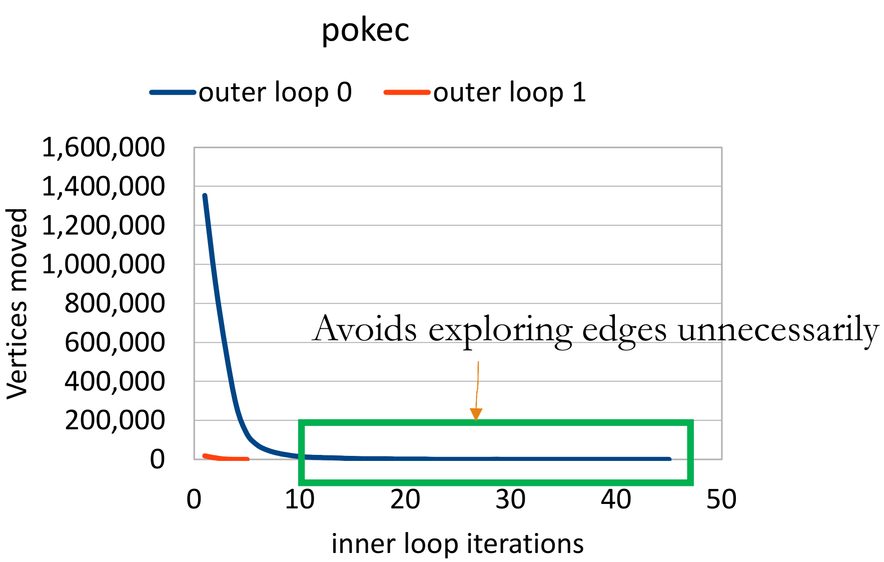
Algorithm 1: Sequential Louvain Algorithm.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

```
//Outer Loop
while true do
    changes=0
    C = initialize each vertex in its own community
    Q = computeModularity(V, E, C)
    //Phase 1 (Inner loop): Modularity Optimization
    while true do
        Qprev = Q
        for all u ∈ V do
            //build NCW
            NCW_u = ⟨c, w_{u→c}⟩ ∀ c ∈ NC_u
            (bestcomm, bestGain) =
              arg max_{c∈NCW_u} ΔQ_{u→c}
            if bestGain>0 and bestcomm ≠ oldcomm of u
              then
                Move u to bestcomm and update C
        // new modularity value
        Q = computeModularity(V, E, C)
        if Q − Qprev ≤ τ then
            break
        changes=1
    //Phase 2: Aggregation and Graph Reconstruction
    if changes = 0 then
        break
    reconstructGraph(V, E, C)
```

Push – Pros

Scans through all neighbors of a vertex only when a vertex changes its community to update NCW



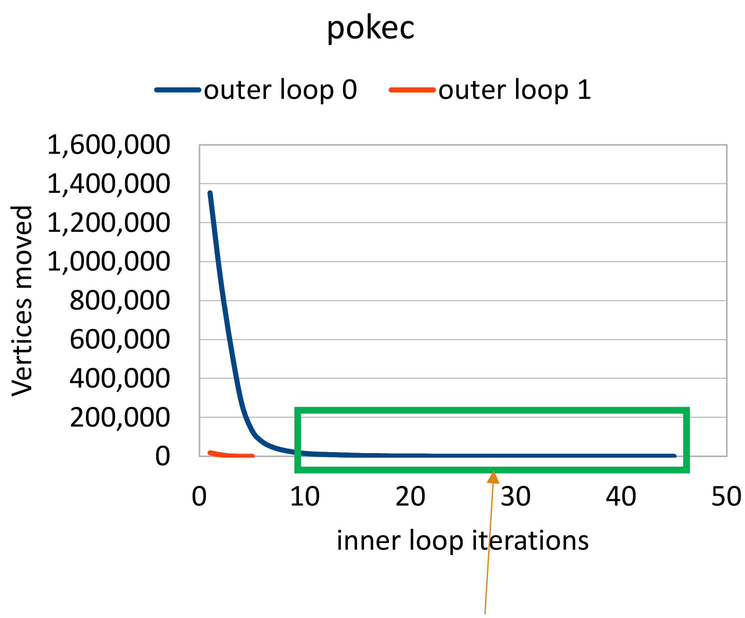
Algorithm 2: A push-based sequential Louvain algorithm.

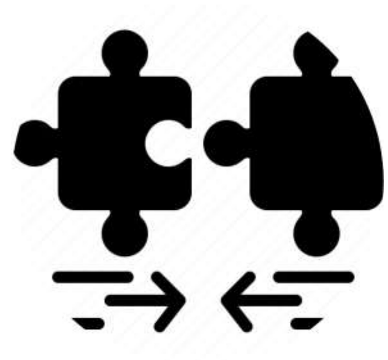
Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Outer Loop
while *true* **do**
  changes=0 C = initialize each vertex in its own community
  Q = computeModularity(V, E, C)
  //build initial NCW_u for all vertices
  $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \; c \in NC_u \; \forall \; c \in V$
  //Phase 1 (Inner loop): Modularity Optimization;
  while *true* **do**
    Qprev = Q
    for *all* $u \in V$ **do**
      (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
      if *bestGain>0 and bestcomm ≠ oldcomm of u* **then**
        Move u to bestcomm and update C
        Update NCW_u for bestcomm and oldcomm
        Update $NCW_x \; \forall \; x \in N_u$, for bestcomm and oldcomm
    // new modularity value
    Q = computeModularity(V, E, C)
    if $Q - Qprev \leq \tau$ **then**
      break
    changes=1
  //Phase 2: Aggregation and Graph Reconstruction
  if *changes = 0* **then**
    break
  reconstructGraph(V, E, C)

Implications

A push-based Louvain algorithm is likely to do fewer edge explorations compared to a pull-based

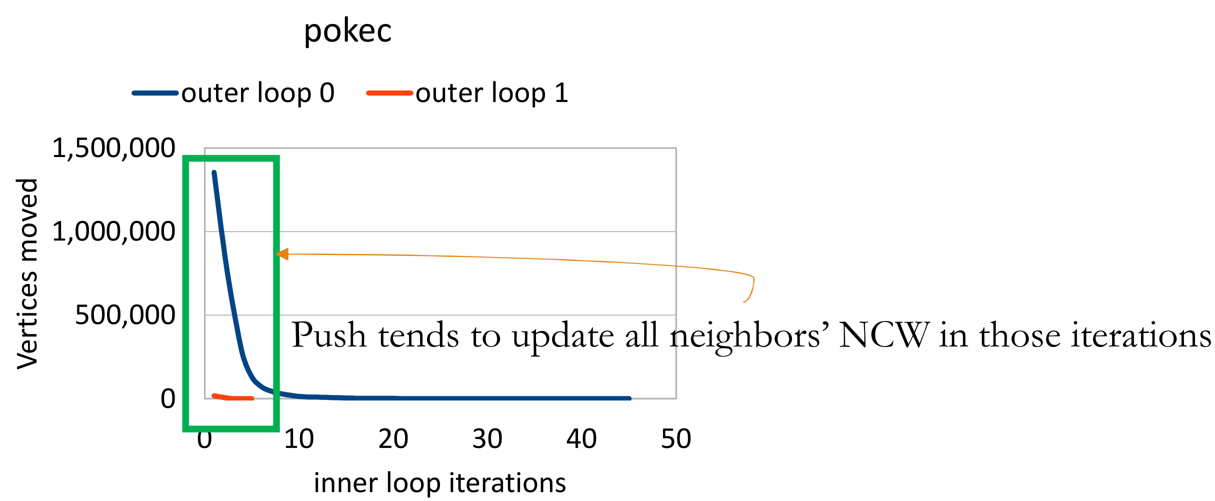




in the later inner loop iterations

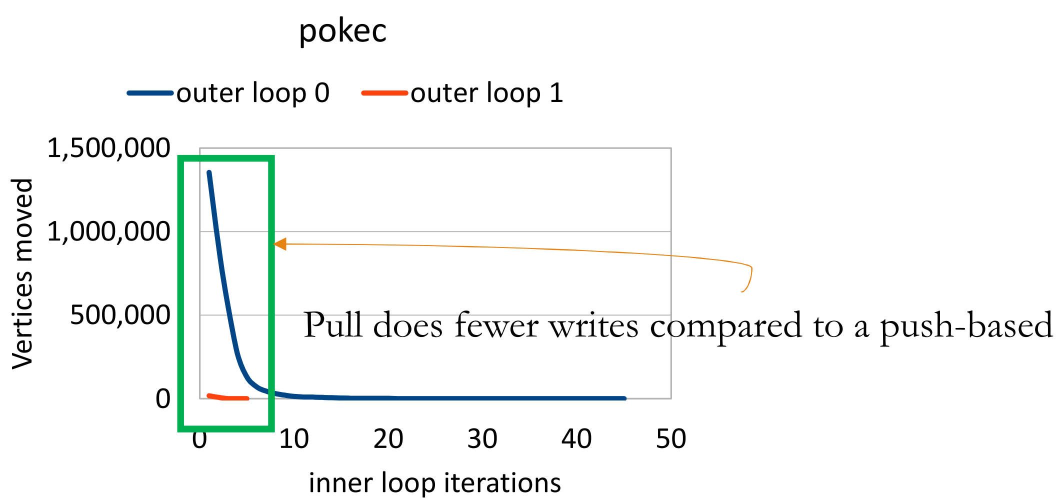
Push – Cons

Push does more writes to memory compared to a pull-based when there is a lot of moves



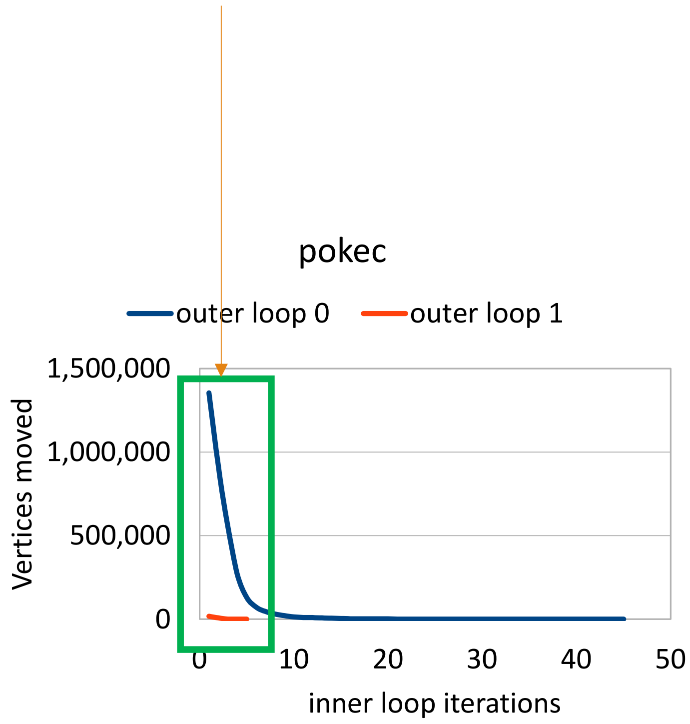
Pull – Pros

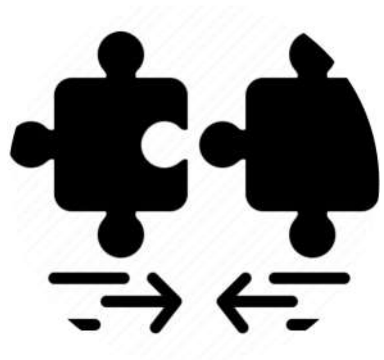
Pull **does fewer writes** compared to a push-based algorithm when there is a lot of moves



Implications

Using a push-based algorithm in the first few inner loop iterations **might not be beneficial**





Take-home Message

Neither pull nor push performs best across all iteration space

Pull-Push/Hybrid Louvain

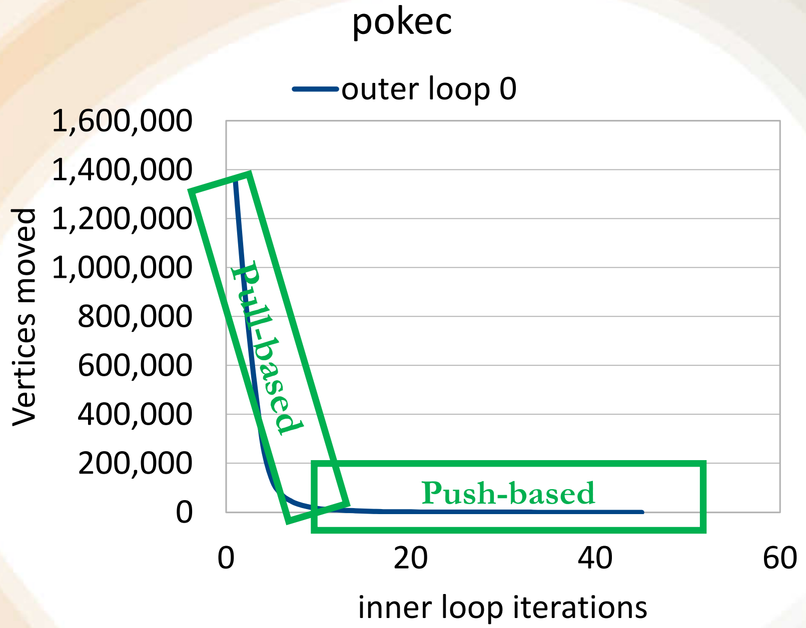
Best of both worlds

Pull-Push Louvain Algorithm

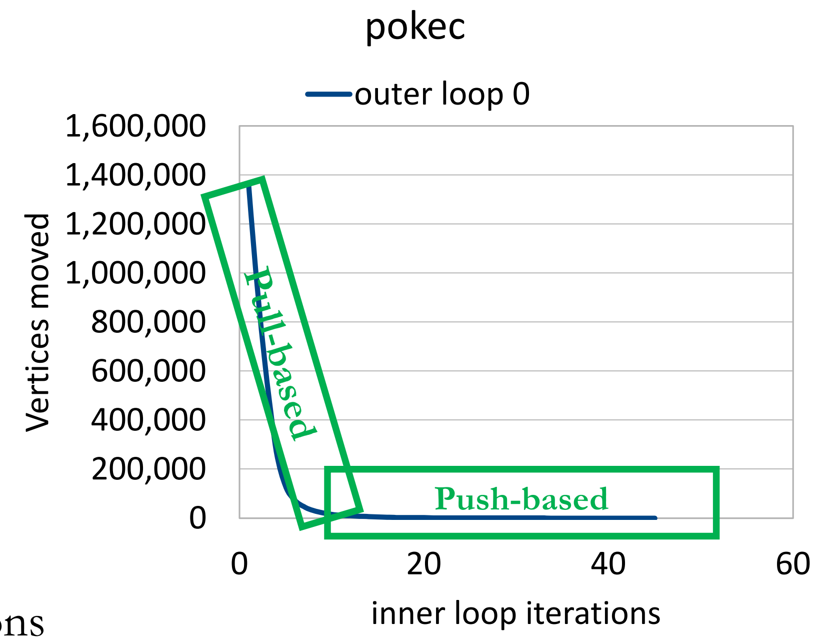
How it works

For a given outer loop

- Start with a pull-based
- Switch to a push-based after a given #of iterations



Pull-Push Louvain Algorithm



Benefits

- Explores a vertex's neighborhood when there a change
- Automatically prunes a significant number of edge-explorations

Automatic Edge Pruning of Pull-Push Algorithm

Graph: POKEC

Algorithmic Improvement		pull	hybrid
Vertices	Visited	833M	833M
	Reduction	**1.00x**	**1.00x**
Edges	Visited	2.34G	0.18G
	Reduction	**1.00x**	**12.82x**

Graph: Hollywood

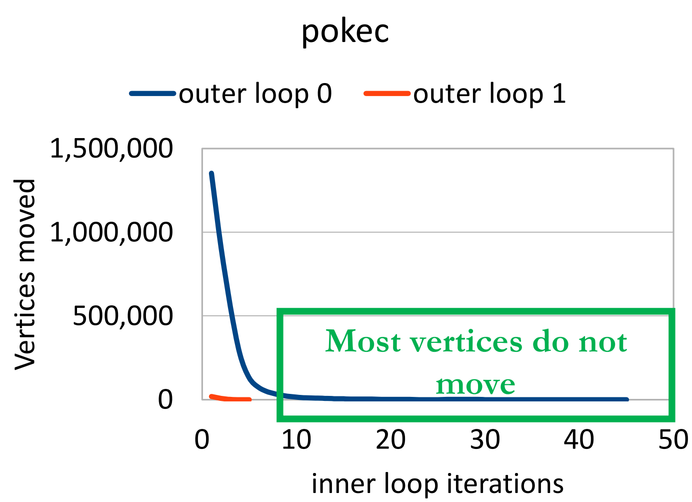
Algorithmic Improvement		pull	hybrid
Vertices	Visited	27.7M	27.7M
	Reduction	**1.00x**	**1.00x**
Edges	Visited	2.82G	0.45G
	Reduction	**1.00x**	**6.20x**

Prunes **6-13 × edges** compared to a standard (pull-based) Louvain

Vertex Pruning

Vertex Pruning

- It is unnecessary to iterate over all vertices - number of vertices changing community drops significantly after the first few inner loop iterations



- We show analytical and intuitive derivation of the vertices that can be pruned with minimal sacrifice

Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

$$Q_{u\to c} \quad = \left[\frac{w_{u\to c}}{2m} - \frac{\sum_{tot}^{c} w(u)}{2m^2}\right] = \frac{1}{2m}\left[w_{u\to c} - \frac{\sum_{tot}^{c} w(u)}{m}\right]$$

Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

$$Q_{u\rightarrow c} = \left[\frac{w_{u\rightarrow c}}{2m} - \frac{\sum_{tot}^{c} w(u)}{2m^2}\right] = \frac{1}{2m}\left[w_{u\rightarrow c} - \frac{\sum_{tot}^{c} w(u)}{m}\right]$$

$w_{u\rightarrow c}$ = sum of edge weights from u to community c

$$\Sigma_{tot}^{c} = \sum W_{u,v}\ , for\ all\ u \in c\ or\ v \in c, m = \sum_{e(u,v)} W_{u,v}$$

Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

$$Q_{u\to c} = \left[\frac{w_{u\to c}}{2m} - \frac{\sum_{tot}^{c} w(u)}{2m^2}\right] = \frac{1}{2m}\left[w_{u\to c} - \frac{\sum_{tot}^{c} w(u)}{m}\right]$$

$$\text{Let, } Cm = \frac{\sum_{tot}^{c}}{m}, Cm = (0, 1\rangle$$

Total graph edge

Total community edge

Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

$$Q_{u \to c} = \left[\frac{w_{u \to c}}{2m} - \frac{\sum_{tot}^{c} w(u)}{2m^2} \right] = \frac{1}{2m} \left[w_{u \to c} - \frac{\sum_{tot}^{c} w(u)}{m} \right]$$

$$if, Cw = Cm * w(u) \Rightarrow Q_{u \to c} \sim w_{u \to c} - Cw$$

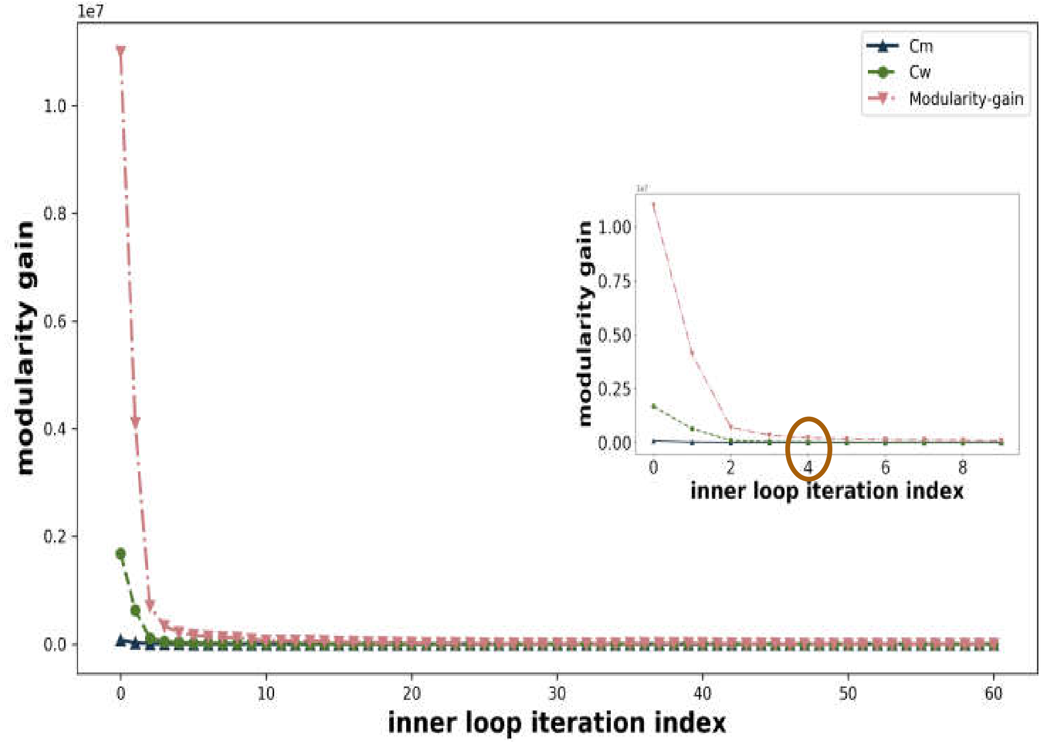
total edges of vertex u

Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

Let, $Cm = \frac{\sum_{tot}^{c}}{m}$, $Cm = (0, 1\rangle$

$$Cw = Cm * w(u) \Rightarrow \boxed{Q_{u \to c} \sim w_{u \to c} - Cw}$$



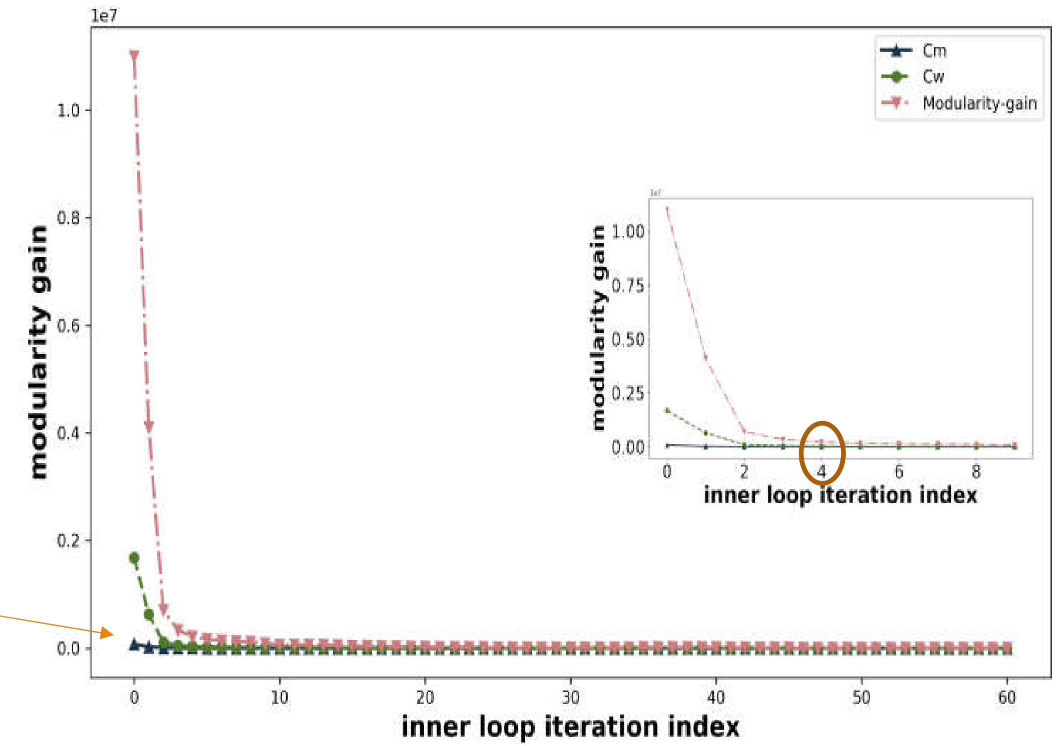
Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

Let, $Cm = \frac{\Sigma_{tot}^{c}}{m}, Cm = (0, 1\rangle$

$Cw = Cm * w(u) \Rightarrow Q_{u \to c} \sim w_{u \to c} - Cw$

Cm does not play an important role in the modularity gain



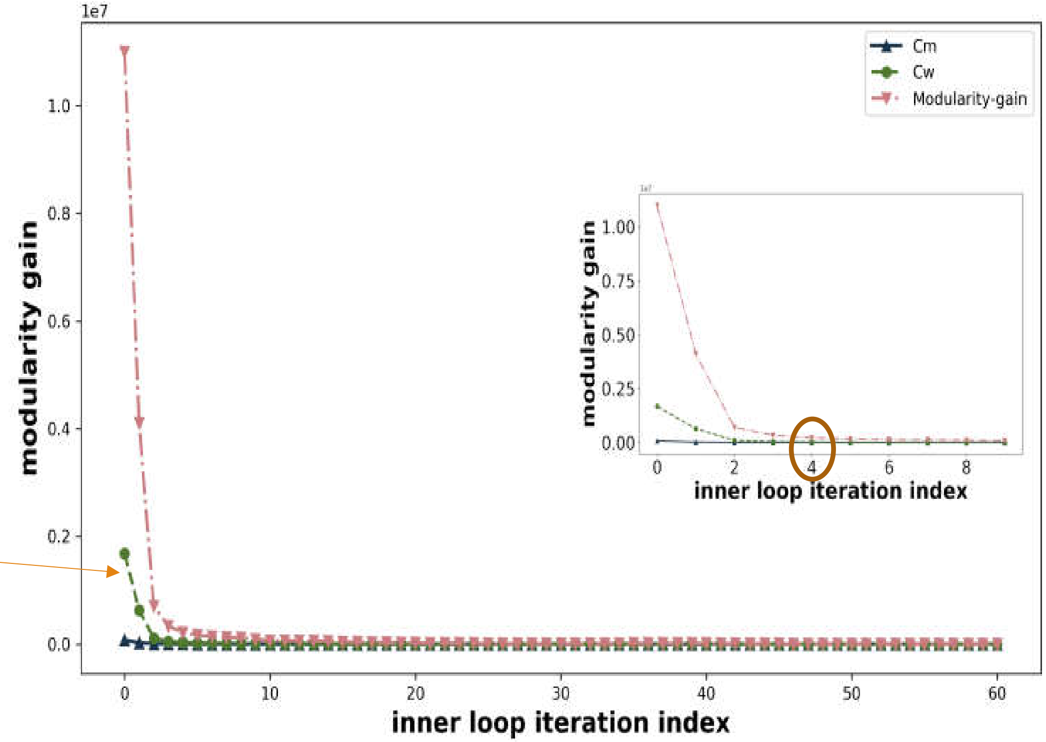
Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

Let, $Cm = \frac{\Sigma_{tot}^{c}}{m}$, $Cm = (0, 1\rangle$

$Cw = Cm * w(u) \Rightarrow Q_{u \to c} \sim w_{u \to c} - Cw$

After first few iterations, Cw does not play an important role in the modularity gain



intel

Vertex Pruning: Analytical Derivation

Modularity gain by moving a vertex u to a community c:

Let, $Cm = \frac{\sum_{tot}^{c}}{m}$, $Cm = (0, 1\rangle$

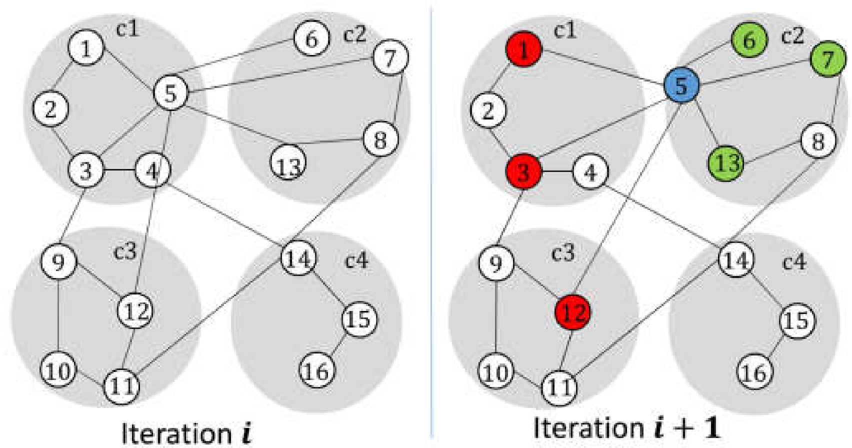
$$Cw = Cm * w(u) \Rightarrow Q_{u \to c} \sim w_{u \to c} - Cw$$

Intuitive: Focus on the vertices whose $w_{u \to c}$ decreases

Intuitive: Skip the vertices whose $\sum_{tot}^{c}$ is impacted by a move

Impact on modularity is small if applied on the push-phases – later inner loop iterations

Vertex Pruning: Analytical Derivation



What to recompute? (Analytical Derivation in Paper)

If a vertex moves, only recompute for its first level neighbors that are ***not*** in its new community
=> recompute **red** neighbors, impacts on green and blue are minimal, no impact on white

Algorithms and Impact of Vertex & Edge Pruning

Algorithm name	What is does
Pull	Standard pull-based Louvain
Pull-prune	Pull + vertex pruning in all iterations
Hybrid	Switching between pull and push
Hybrid-prune	Hybrid + **vertex pruning in push phases only**

Graph: POKEC

Algorithmic Improvement		pull	pull prune	hybrid	hybrid prune
Vertices	Visited	833M	6.09M	833M	9.49M
	Reduction	**1.00x**	**13.68x**	**1.00x**	**8.78x**
Edges	Visited	2.34G	0.3G	0.18G	0.182G
	Reduction	**1.00x**	**7.70x**	**12.82x**	**12.82x**

Graph: Hollywood

Algorithmic Improvement		pull	pull prune	hybrid	hybrid prune
Vertices	Visited	27.7M	5.44M	27.7M	6.12M
	Reduction	**1.00x**	**5.09x**	**1.00x**	**4.52x**
Edges	Visited	2.82G	0.95G	0.45G	0.45G
	Reduction	**1.00x**	**2.98x**	**6.20x**	**6.20x**

- Prune 4 to 12× vertices

Algorithms and Impact of Vertex & Edge Pruning

Algorithm name	What is does
Pull	Standard pull-based Louvain
Pull-prune	Pull + vertex pruning in all iterations
Hybrid	Switching between pull and push
Hybrid-prune	Hybrid + **vertex pruning in push phases only**

Graph: POKEC

Algorithmic Improvement		pull	pull prune	hybrid	hybrid prune
Vertices	Visited	833M	6.09M	833M	9.49M
	Reduction	**1.00x**	**13.68x**	**1.00x**	**8.78x**
Edges	Visited	2.34G	0.3G	0.18G	0.182G
	Reduction	**1.00x**	**7.70x**	**12.82x**	**12.82x**

Graph: Hollywood

Algorithmic Improvement		pull	pull prune	hybrid	hybrid prune
Vertices	Visited	27.7M	5.44M	27.7M	6.12M
	Reduction	**1.00x**	**5.09x**	**1.00x**	**4.52x**
Edges	Visited	2.82G	0.95G	0.45G	0.45G
	Reduction	**1.00x**	**2.98x**	**6.20x**	**6.20x**

- **Does not prune additional edges compared to hybrid**

Performance Benefit using Single Thread

Graphs	Pull		Hybrid		Pull-Prune		Hybrid-Prune	
	Q	T	Q	T	Q	T	Q	T
Wikipedia	0.57	98.9	0.57	74.1	0.57	65.3	0.57	61.9
Hollywood	0.73	57.2	0.73	14.8	0.73	31.5	0.73	12.9
POKEC	0.68	19.3	0.68	11.1	0.68	4.82	0.68	5.7

Q= Modularity, T= Time (s)

- Edge pruning : 1.3× – 3.9×
- Vertex pruning : 1.5× – 4 ×
- Vertex pruning on top of edge pruning: upto1.9×

Take-home Message

Even without any parallelization, edge and vertex pruning gives up to 4x speedup over the standard Louvain algorithm.

Parallel
Pull, Push, Pull-Push Algorithms

Parallel Pull-based Louvain

Private hashmap for each thread

Algorithm 3: Parallel Pull-based inner loop of Louvain.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Phase 1 (Inner loop): Modularity Optimization
while *true* **do**
    Qprev = Q
    Parallel for *all* $u \in V$ **do**
        //build NCW
        $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \; c \in NC_u$
        (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
        if *bestGain>0 and bestcomm* $\neq$ *oldcomm of u* **then**
            Move u to bestcomm and update C **atomically**
    //new modularity value
    Q = **Parallel** computeModularity(V, E, C)
    if $Q - Qprev \leq \tau$ **then**
        break
    changes=1

Parallel Pull-based Louvain

For each vertex in parallel

Algorithm 3: Parallel Pull-based inner loop of Louvain.

Data: Graph $G = (V, E)$, $\tau = Threshold$.

Result: Final Community Assignment C, and Modularity Q

//Phase 1 (Inner loop): Modularity Optimization

while *true* **do**

  Qprev = Q

  Parallel for *all* $u \in V$ **do**

    //build NCW

    $NCW_u = \langle c, w_{u \to c} \rangle \ \forall \ c \in NC_u$

    (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$

    if *bestGain>0 and bestcomm* $\neq$ *oldcomm of* u **then**

      Move u to bestcomm and update C **atomically**

  //new modularity value

  Q = **Parallel** computeModularity(V, E, C)

  if $Q - Qprev \leq \tau$ **then**

    break

  changes=1

Parallel Pull-based Louvain

Change community membership atomically

Algorithm 3: Parallel Pull-based inner loop of Louvain.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Phase 1 (Inner loop): Modularity Optimization
while *true* **do**
    Qprev = Q
    Parallel for *all* $u \in V$ **do**
        //build NCW
        $NCW_u = \langle c, w_{u \to c} \rangle \; \forall \; c \in NC_u$
        (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
        if *bestGain>0 and bestcomm* $\neq$ *oldcomm of* u **then**
            Move u to bestcomm and update C **atomically**
    //new modularity value
    Q = **Parallel** computeModularity(V, E, C)
    if $Q - Qprev \leq \tau$ **then**
        break
    changes=1

Parallel Pull-based Louvain

Compute modularity using parallel reduction

Algorithm 3: Parallel Pull-based inner loop of Louvain.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//Phase 1 (Inner loop): Modularity Optimization
while *true* **do**
    Qprev = Q
    Parallel for *all* $u \in V$ **do**
        //build NCW
        $NCW_u = \langle c, w_{u \to c} \rangle \ \forall \ c \in NC_u$
        (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q_{u \to c}$
        if *bestGain>0 and bestcomm ≠ oldcomm of* u **then**
            Move u to bestcomm and update C **atomically**
    //new modularity value
    Q = **Parallel** computeModularity(V, E, C)
    if $Q - Qprev \leq \tau$ **then**
        break
    changes=1

Parallel Push-based

Shared hashmap of size O(E)

Update hashmaps using Locks

Algorithm 4: Parallel Push-based inner loop of Louvain.

Data: Graph $G = (V, E)$, $\tau = Threshold$.
Result: Final Community Assignment C, and Modularity Q

//build initial NCW_u for all vertices
Parallel $\forall\, u \in V$ do $NCW_u = \langle c, w_{u \to c} \rangle\ \forall\, c \in NC_u$
//Phase 1 (Inner loop): Modularity Optimization
while *true* **do**
  Qprev = Q
  Parallel for *all* $u \in V$ **do**
    (bestcomm, bestGain) = $arg\ max_{c \in NCW_u} \Delta Q(u)$ moving to c)
    if *bestGain>0 and bestcomm* $\neq$ *oldcomm of u* **then**
      Move u to bestcomm and update C **atomically**
      lock NCW_u
      Update NCW_u for bestcomm and oldcomm
      unlock NCW_u
      lock NCW_x
      Update $NCW_x\ \forall\, x \in N_u$, for bestcomm and oldcomm
      unlock NCW_x
  //new modularity value
  Q = **Parallel** computeModularity(V, E, C)
  if $Q - Qprev \leq \tau$ **then**
    break
  changes=1

Experimental Results

Graphs	V	E	Graphs	V	E
CA	1.08E+05	1.87E+05	CitationCiteseer	2.68E+05	2.31E+06
CaidaRouterLevel	1.92E+05	1.22E+06	CoAuthorsDBLP	2.99E+05	1.96E+06
POKEC	5.40E+05	3.05E+07	CoPapersCiteseer	4.34E+05	3.21E+07
Hollywood	1.14E+06	1.13E+08	Amazon	5.49E+05	1.85E+06
Wikipedia	3.97E+07	9.01E+07	As-Skitter	1.70E+06	2.22E+07
Uk-2005	1.68E+07	3.96E+08	Rgg_n_2_24_s0	1.68E+07	2.65E+08
Friendster	6.65E+07	1.89E+09	Webbase-2001	1.18E+08	1.02E+09

Input Graphs

Performance Analysis Platform

Experimental Platforms

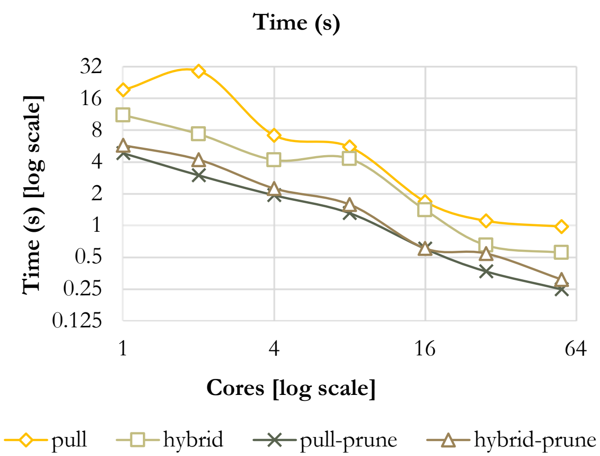
Platform Metric	Platform 1	Platform 2
Processor	Intel(R) Xeon(R) Platinum 8180	Intel(R) Xeon(R) CPU E7-8880 v3
CPU Clock	2.50GHz	2.30GHz
Sockets	2	4
Cores	56 (each socket has 28)	72 (each socket with 18 cores)
L3 Cache	97 MB	46.1 MB
Memory Speed	2666 MHz	1200 MHz
Memory Size	196.7GB	1 TB
Compiler	Intel ICC 18.0	
Parallel Program	C with OpenMP	

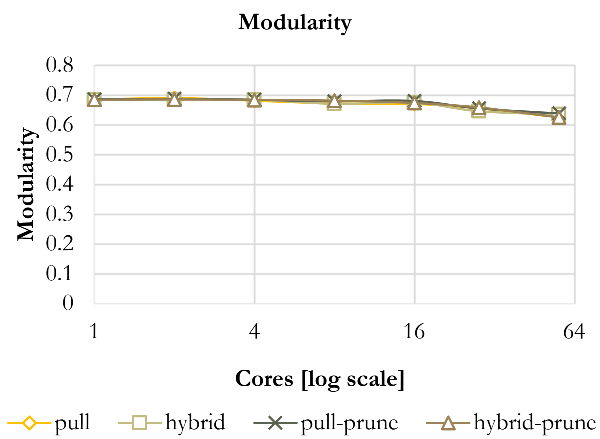
Algorithms

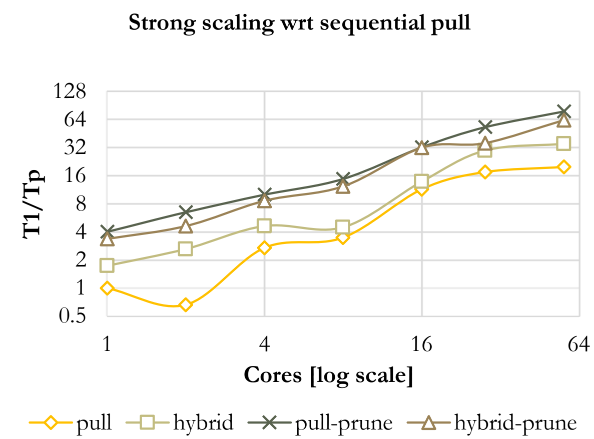
Algorithm name	What is does
Pull	Standard pull-based Louvain
Pull-prune	Pull + vertex pruning in all iterations
Hybrid	Switching between pull and push
Hybrid-prune	Hybrid + **vertex pruning in push phases only**

Hybrid Pull-Push vs Pull Based Louvain

Dataset: POKEC, Outer loop 0







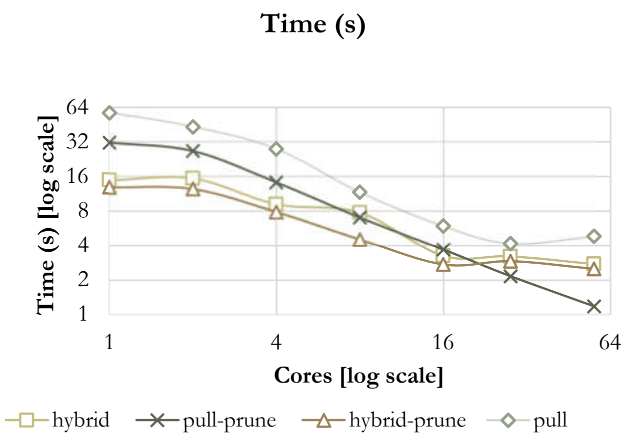
On the 56 cores of Skylake

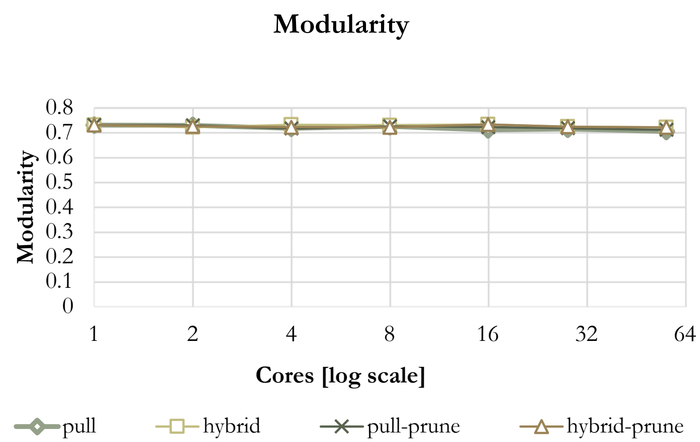
- Pull algorithm gets 19.8×
- Pull-prune gets 78×
- Hybrid gets 35×
- Hybrid-prune gets 63× speedup

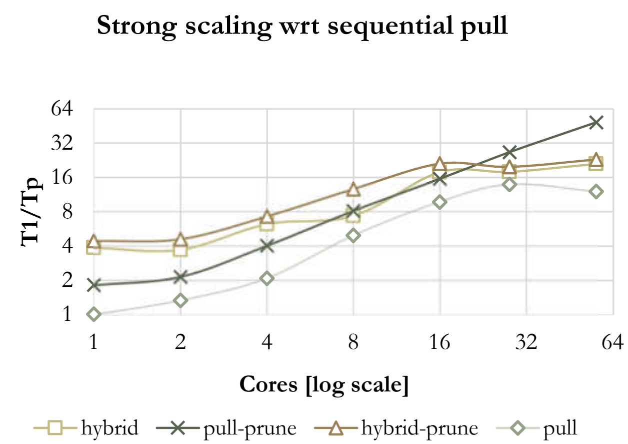
Graphs	V	E
POKEC	**5.40E+05**	**3.05E+07**

Hybrid Pull-Push vs Pull Based Louvain

Dataset: Hollywood, Outer loop 0







On the 56 cores of Skylake

- Pull algorithm gets 12×
- Pull-prune gets 26×
- Hybrid gets 21×
- Hybrid-prune gets 23× speedup

Graphs	V	E
Hollywood	1.14E+06	1.13E+08

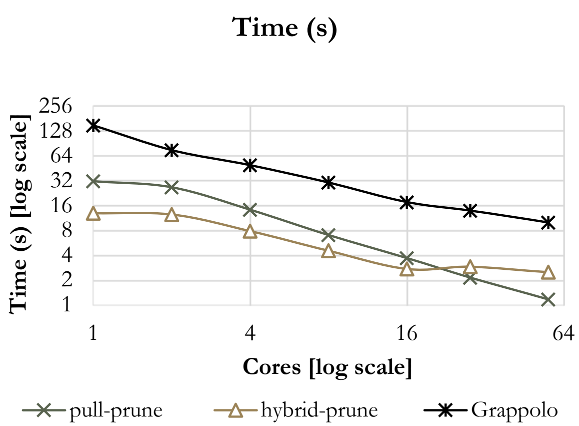
Comparison with Prior State-of-the-art

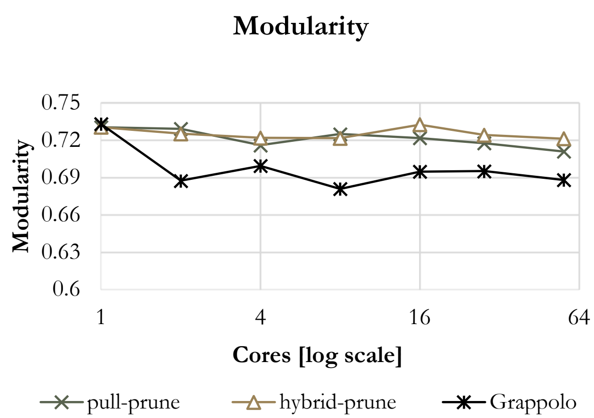
The Louvain in Grappolo

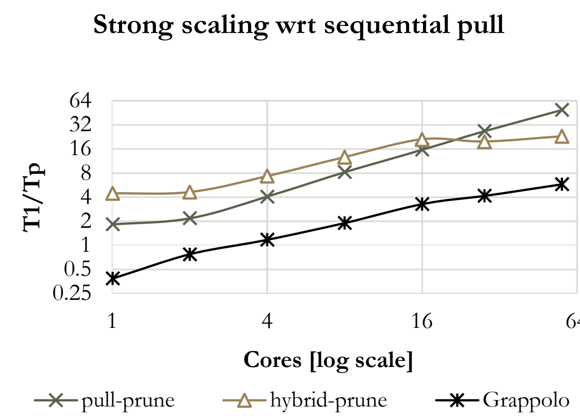
Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel Comput.* 47 (2015), 19–37

5-11 × faster than the Louvain in Grappolo

Dataset: Hollywood, Outer loop 0







Compared to Grappolo

- Pull-prune 5-11 × faster
- Hybrid-prune 5-8 × faster, **best modularity**
- Modularity is higher in pull-prune and hybrid-prune

Graphs	V	E
Hollywood	**1.14E+06**	**1.13E+08**

2-8 × faster than the Louvain in Grappolo

Graph	hybrid-prune		pull-prune		Grappolo		Grappolo vs hybrid-prune	
	Q	T	Q	T	Q	T	Modularity	Speedup
caidaRouterLevel	0.68	0.05	0.65	0.02	0.68	0.11	1.00	2.20
citationCiteeer	0.6	0.06	0.62	0.04	0.59	0.3	1.02	5.00
coPaperDBLP	0.77	0.41	0.77	0.11	0.71	0.85	1.08	2.07
coPaperCiteeer	0.84	0.37	0.84	0.12	0.8	0.85	1.05	2.30
as-Skitter	0.72	0.9	0.71	1.37	0.69	2.12	1.04	2.36
uk-2005	0.95	21.01	0.88	17.71	0.83	136.95	1.14	6.52
rgg_n_2_24_0	0.92	1.71	0.89	1.75	0.74	13.54	1.24	7.92

Q= Modularity, T= Time (s)

The higher the better

Compared to Grappolo (on 56 cores of Skylake)

- Pull-prune is 2- 8 × faster
- Hybrid-prune is 2 - 8 × faster, provides **best modularity**

skyLake Core	Graph	Grappolo		Pull		Hybrid		Pull-prune		Hybrid-prune		Speedup
		Q	T	Q	T	Q	T	Q	T	Q	T	
1	amazon	0.67	3.76	0.69	0.64	0.69	0.69	0.68	0.23	0.69	0.53	16.05
8		0.67	0.79	0.68	0.12	0.68	0.11	0.68	0.09	0.68	0.08	9.49
1	ca	0.54	0.30	0.56	0.10	0.56	0.09	0.56	0.04	0.56	0.07	4.22
8		0.54	0.08	0.56	0.02	0.56	0.01	0.56	0.01	0.56	0.01	8.21

Q= Modularity, T= Time (s)

The higher the better

Compared to Grappolo (on 56 cores of Skylake)

- Hybrid-prune is 4-16 × faster
- Modularity is always higher or the same

4-16 × faster than the Louvain in Grappolo

Quality: Normalized Mutual Information (NMI)

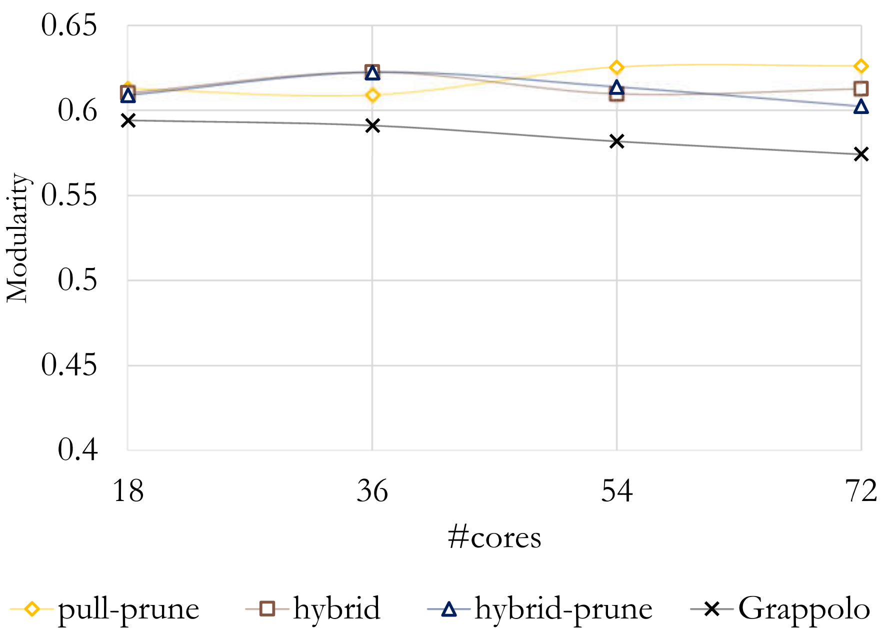
NMI score >0.8 is considered good

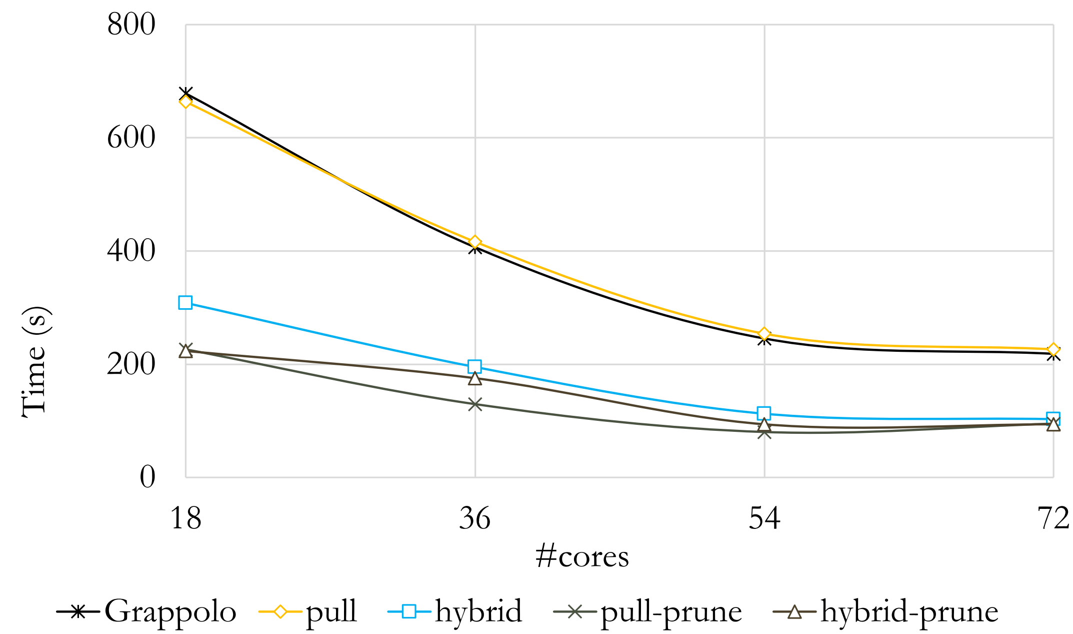
Algorithm		Pull		Pull_prune		Hybrid		Hybrid_prune		Grappolo	
Threads		1	56	1	56	1	56	1	56	1	56
NMI Score	ca	1.000	0.995	1.000	0.995	0.999	0.995	1.000	0.995	0.996	0.980
	amazon	1.000	0.991	0.999	0.990	0.998	0.991	0.999	0.990	0.991	0.946

Our algorithms are better in NMI score than Grappolo, baseline is sequential Louvain Algorithm

Louvain on Large Graphs

FRIENDSTER, V = 65,608,366 E = 3,612,134,270





Compared to Grappolo (on 72 cores of Haswell)

- Pull-prune and Hybrid-prune is 2-4x faster
- Better Modularity

Platform: 72 core Haswell machine

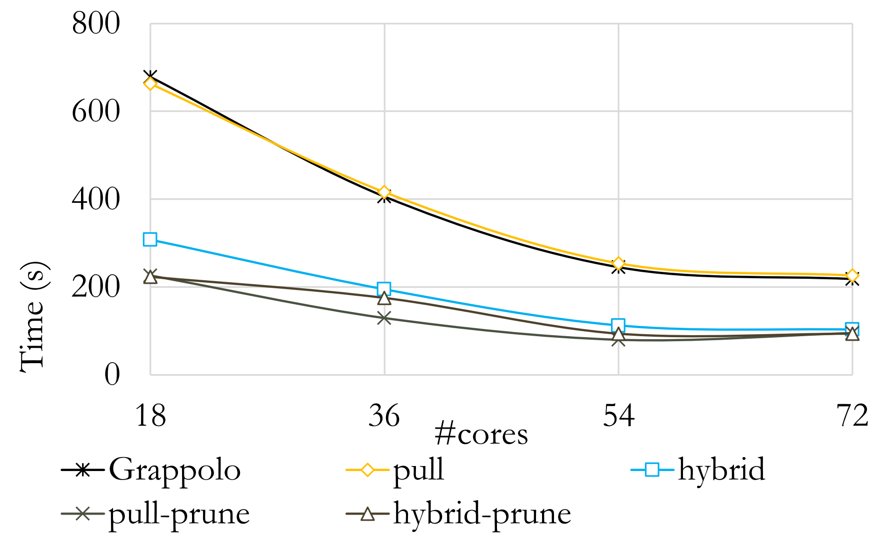
Comparison with Recent Distributed Memory Algorithm

"Our MPI+OpenMP implementation yields about 7x speedup (on 4K processes) for soc-friendster network (1.8B edges) over Grappolo on 64 threads on NERSC CORI system), without compromising output quality"

TABLE III
DISTRIBUTED MEMORY VS SHARED MEMORY (GRAPPOLO) PERFORMANCE (RUNTIME) OF LOUVAIN ALGORITHM ON A SINGLE CORI NODE USING 4-64 THREADS. THE INPUT GRAPH IS SOC-FRIENDSTER (1.8B EDGES).

#Threads	Distributed memory (sec.)	Shared memory (sec.)
4	6,082.25	1,216.54
8	3,615.52	843.37
16	2,252.09	725.26
32	1,515.24	689.38
64	1,303.98	554.52

2.3x



Sayan et. al. ***Distributed Louvain Algorithm for Graph Community Detection,*** *IPDPS 2018*

Comparison with Recent Distributed Memory Algorithm

"Our MPI+OpenMP implementation yields about 7x speedup (on 4K processes) for soc-friendster network (1.8B edges) over Grappolo on 64 threads on NERSC CORI system), without compromising output quality"

TABLE III

DISTRIBUTED MEMORY VS SHARED MEMORY (GRAPPOLO) PERFORMANCE (RUNTIME) OF LOUVAIN ALGORITHM ON A SINGLE CORI NODE USING 4-64 THREADS. THE INPUT GRAPH IS SOC-FRIENDSTER (1.8B EDGES).

#Threads	Distributed memory (sec.)	Shared memory (sec.)
4	6,082.25	1,216.54
8	3,615.52	843.37
16	2,252.09	725.26
32	1,515.24	689.38
64	1,303.98	554.52

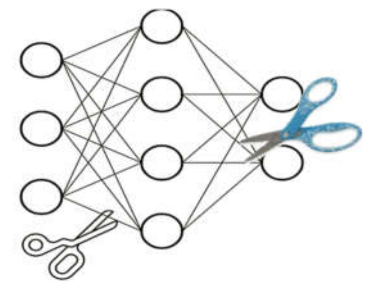
2.3x

Quick math says our approach could be 4 - 8x faster than this algorithm

Sayan et. al. ***Distributed Louvain Algorithm for Graph Community Detection,*** *IPDPS 2018*

Conclusion – a new state-of-art for Louvain

- Prune unnecessary edge and vertex exploration during community detection
- Edges pruned by 6 to 13× without sacrificing quality – up to 4x speedup
- Vertex pruned by 4 to 12× with minimal sacrifice quality – up to 4x speedup
- Parallel algorithms 2-16x faster than prior state-of-the-art without sacrificing quality



We will be happy to make the code public. Please contact: jesmin.jahan.tithi@intel.com