Distributed Processing System: Methodology, Implementation, Application

Dong Liu

Talk topics

- Partition
- Task Scheduling
- Flexibility
- Balance
- Fault Tolerance
- Algorithm Implementation

Graph partition

Graph partition is the first step of graph processing, partition result will directly affect the performance of communication, task scheduling, workload balance and etc.

Graph partition methods can be divided into serval types:

(i) Vertex-cut (static)
(ii) Edge-cut (dynamic)
(iii) Hybrid-cut
(iv) Stream graph cut

Vertex-cut and Edge-cut are most commonly used among all of them.

Challenges to Partitioning

The research challenges of the partitioning fall into two different key issues: quality and scalability

First, high partitioning quality, which is measured by the total number of vertex cuts(edge cuts), is difficult to obtain since the minimization of the vertex cuts is proved to be an NP-hard problem.

Second, the partitioning algorithm is required to scale to deal with large scale graphs.

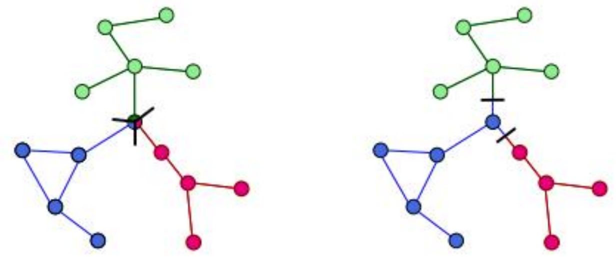
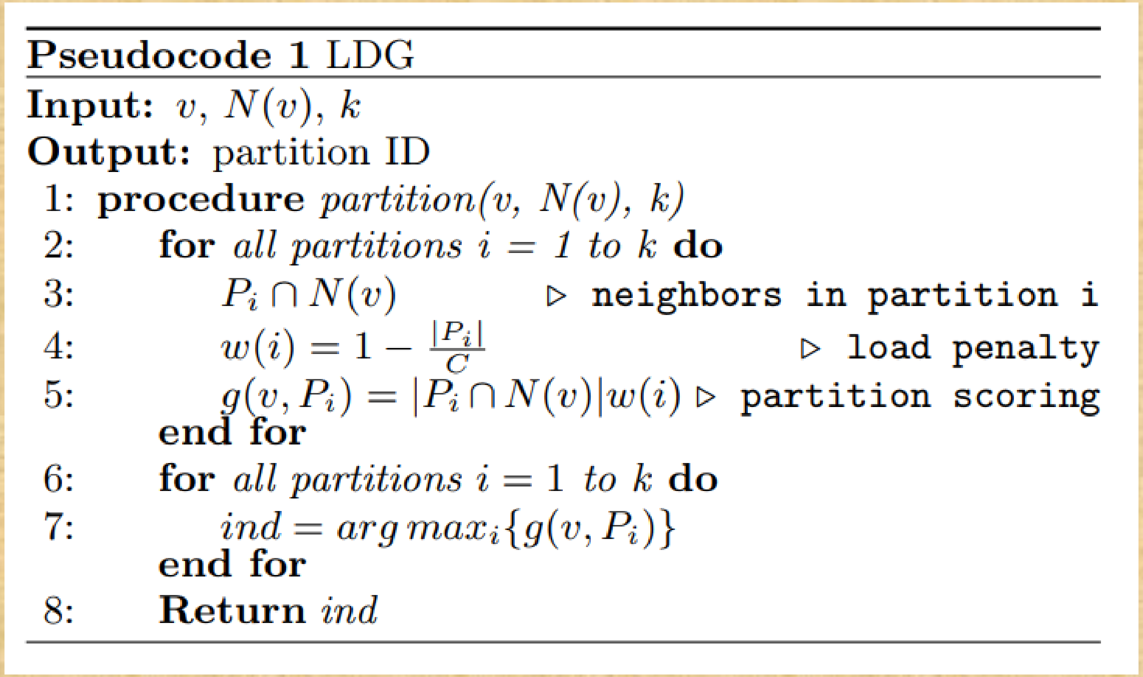


Figure 1: (a) Edge Partition (Vertex-Cut Partition) vs (b) Vertex Partition (Edge-Cut Partition).

The edge partitioning provides the better workload balance because the computational cost of the graph processing essentially depends on the number of edges rather than the number of vertices.

The vertex partitioning causes serious workload imbalance between high-degree vertices and low-degree vertices in the large-scale real-world graph, which mostly has skewed-degree distribution, namely, there are a few high-degree vertices, whereas the rest of the vertices have low degree.

Vertex Partitioning Methods



Pseudocode 1 LDG

Input: v, $N(v)$, k

Output: partition ID

```
1: procedure partition(v, N(v), k)
2:     for all partitions i = 1 to k do
3:         $P_i \cap N(v)$                    ▷ neighbors in partition i
4:         $w(i) = 1 - \frac{|P_i|}{C}$                ▷ load penalty
5:         $g(v, P_i) = |P_i \cap N(v)| w(i)$ ▷ partition scoring
       end for
6:     for all partitions i = 1 to k do
7:         $ind = arg\,max_i\{g(v, P_i)\}$
       end for
8:     Return ind
```

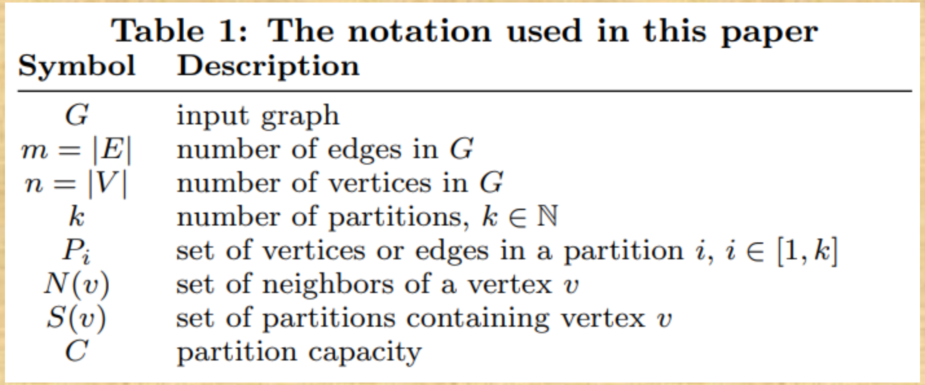


Table 1: The notation used in this paper

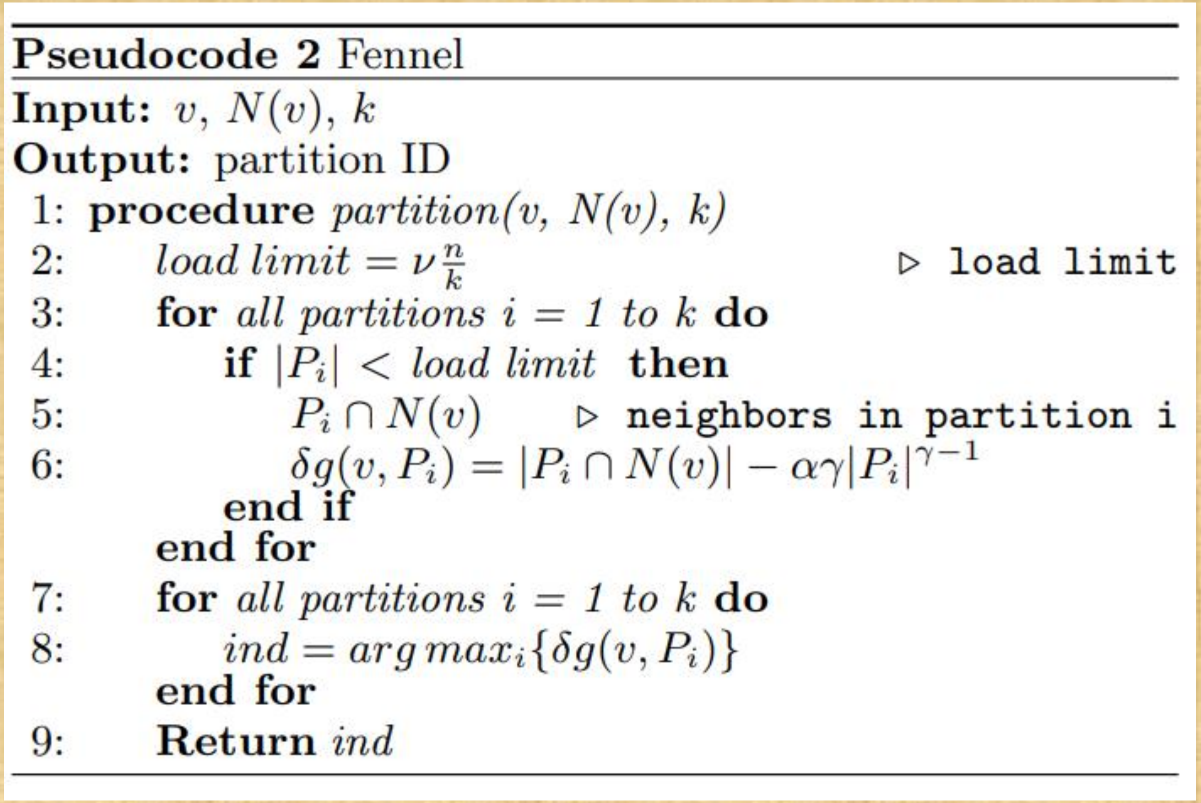
Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity

Linear Deterministic Greedy partitioning (LDG) tries to place neighboring vertices to the same partition, in order to yield fewer edge-cuts. It uses a greedy heuristic that assigns a vertex to the partition containing most of its neighbors while respecting certain capacity constraints.

LDG selects the partition that maximizes $|P_i \cap N(v)|$, the number of neighbors already assigned to a partition while enforcing the capacity constraint $C = \frac{n}{k}$.
The load penalty enforces load balancing to avoid the extreme case where all vertices end up in the same partition.

LDG requires the number of vertices n to be known a priori for calculating the capacity constraint C. Hence, it is generally unsuitable for unbounded processing.

Vertex Partitioning Methods



Pseudocode 2 Fennel

```
Input: $v$, $N(v)$, $k$
Output: partition ID
1: procedure partition($v$, $N(v)$, $k$)
2:     load limit $= \nu \frac{n}{k}$                          ▷ load limit
3:     for all partitions $i$ = 1 to $k$ do
4:         if $|P_i| <$ load limit then
5:             $P_i \cap N(v)$          ▷ neighbors in partition i
6:             $\delta g(v, P_i) = |P_i \cap N(v)| - \alpha\gamma|P_i|^{\gamma-1}$
           end if
       end for
7:     for all partitions $i$ = 1 to $k$ do
8:         $ind = arg\,max_i\{\delta g(v, P_i)\}$
       end for
9:     Return $ind$
```

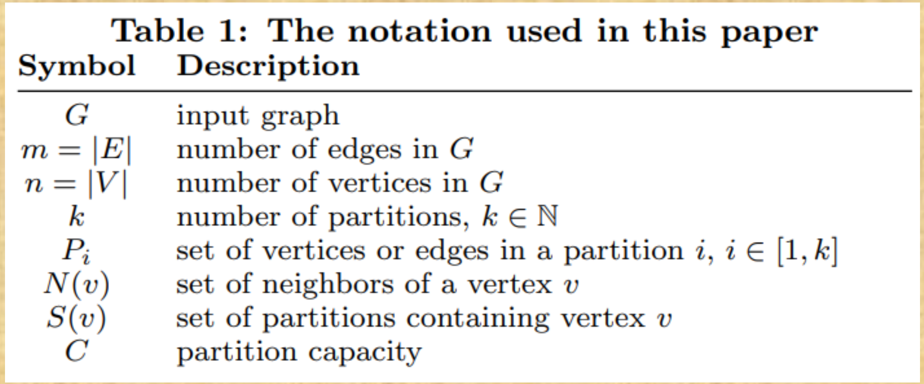


Table 1: The notation used in this paper

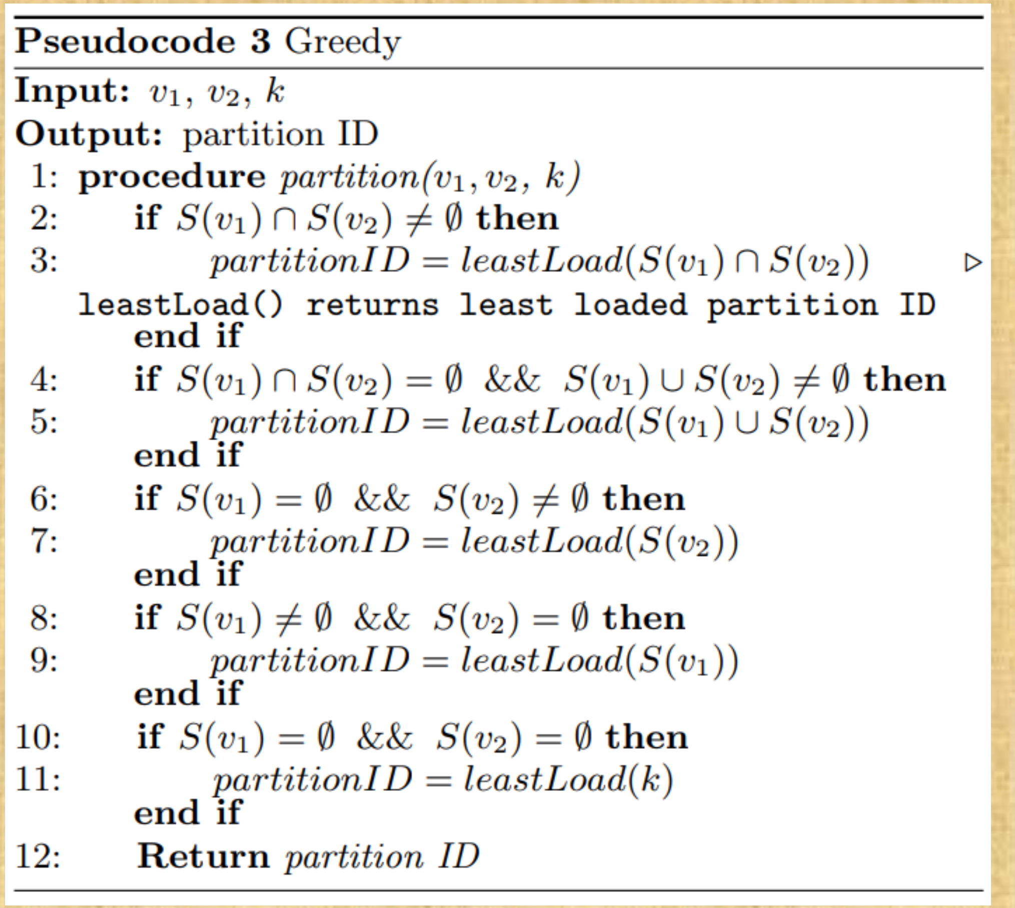
Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity

Fennel is a partitioning strategy whose heuristic combines low edge-cut with balancing goals. Fennel's core idea is to interpolate between maximizing the co-location of neighbouring vertices and minimizing that of non-neighbours. As in LDG, Fennel computes the number of neighbors present in each partition for every input vertex. In addition, the load limit per partition which sets a threshold for the maximum number of assigned vertices.

Here,α $=\sqrt{k}\frac{m}{n^{3/2}}$and the load limit = ν$\frac{n}{k}$. The parameters α, γ, and ν are tunable .In this experiments , γ = 1.5, ν = 1.1.

Similar to LDG, Fennel requires the parameters n and m to be known a priori, and maintaining a persistent state of the assigned partitions during execution, that makes it unsuitable for partitioning unbounded streams.

Edge Partitioning Methods



Pseudocode 3 Greedy

```
Input: $v_1$, $v_2$, $k$
Output: partition ID
 1: procedure $partition(v_1, v_2, k)$
 2:     if $S(v_1) \cap S(v_2) \neq \emptyset$ then
 3:         $partitionID = leastLoad(S(v_1) \cap S(v_2))$    ▷ leastLoad() returns least loaded partition ID
        end if
 4:     if $S(v_1) \cap S(v_2) = \emptyset$ && $S(v_1) \cup S(v_2) \neq \emptyset$ then
 5:         $partitionID = leastLoad(S(v_1) \cup S(v_2))$
        end if
 6:     if $S(v_1) = \emptyset$ && $S(v_2) \neq \emptyset$ then
 7:         $partitionID = leastLoad(S(v_2))$
        end if
 8:     if $S(v_1) \neq \emptyset$ && $S(v_2) = \emptyset$ then
 9:         $partitionID = leastLoad(S(v_1))$
        end if
10:     if $S(v_1) = \emptyset$ && $S(v_2) = \emptyset$ then
11:         $partitionID = leastLoad(k)$
        end if
12:     Return $partition\ ID$
```

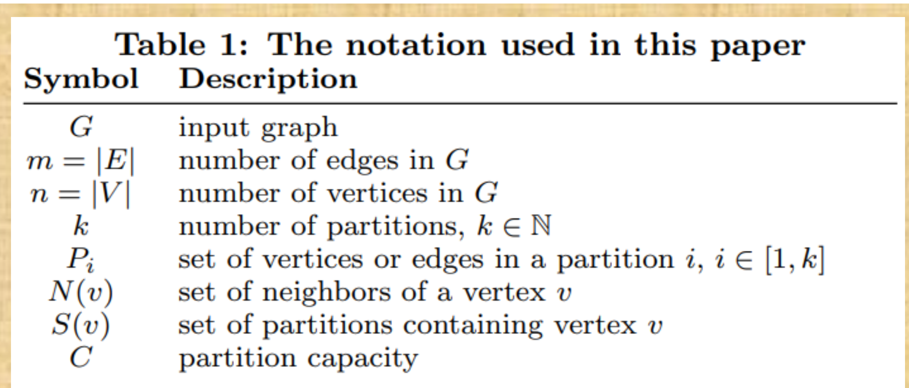


Table 1: The notation used in this paper

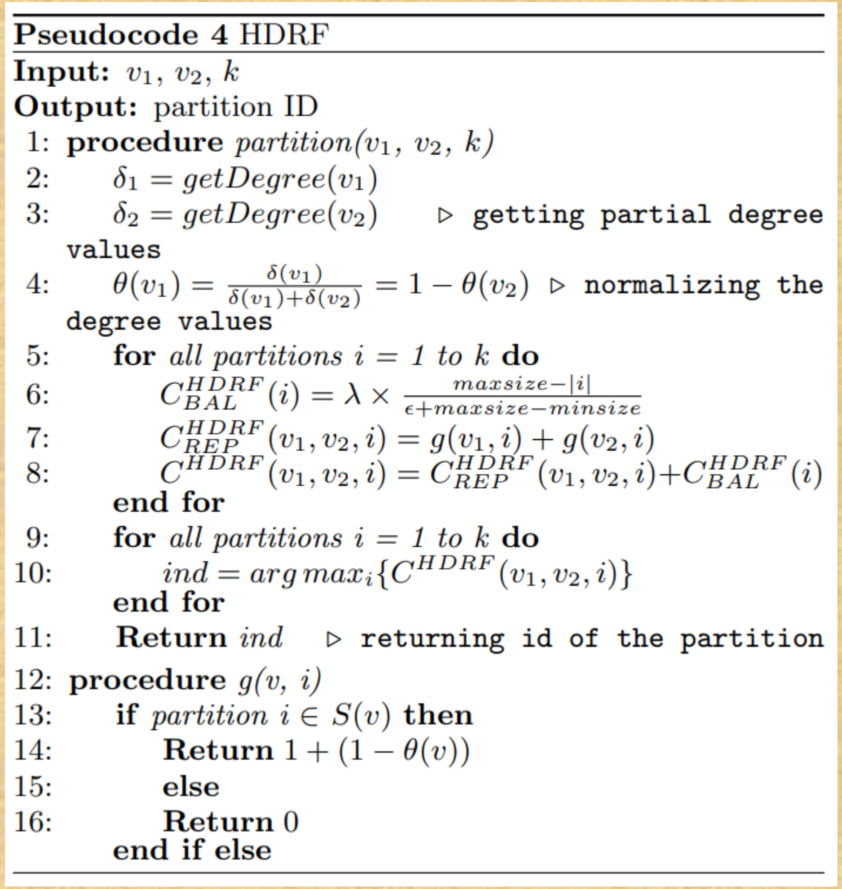
Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity

Greedy aims to minimize vertex-cuts while also assigning balanced load across partitions.
For each edge in the input stream, Greedy examines the participation of the endpoint vertices to existing partitions by applying the following rules:

1) Rule 1: If both endpoint vertices have been previously assigned in any common partition pick the least loaded common partition.
2) Rule 2: If both endpoint vertices have been previously assigned in different partitions pick the least loaded from the union of all assigned partitions.
3) Rule 3: In case either vertex has been previously assigned, pick the least loaded partition from the assigned partitions of that vertex.
4) Rule 4: If none of the vertices has been previously assigned, then pick the least loaded partition overall.

Greedy does not require any knowledge of graph properties before processing the stream. Therefore, it can potentially process an unbounded edge stream. However, it requires maintaining the current partition assignment as a synopsis, which, in case of an unbounded stream would also grow without bound.

Edge Partitioning Methods



Pseudocode 4 HDRF

Input: v_1, v_2, k

Output: partition ID

```
1:  procedure partition($v_1$, $v_2$, $k$)
2:      $\delta_1 = getDegree(v_1)$
3:      $\delta_2 = getDegree(v_2)$      ▷ getting partial degree values
4:      $\theta(v_1) = \frac{\delta(v_1)}{\delta(v_1)+\delta(v_2)} = 1 - \theta(v_2)$   ▷ normalizing the degree values
5:      for all partitions $i$ = 1 to $k$ do
6:          $C_{BAL}^{HDRF}(i) = \lambda \times \frac{maxsize-|i|}{\epsilon+maxsize-minsize}$
7:          $C_{REP}^{HDRF}(v_1, v_2, i) = g(v_1, i) + g(v_2, i)$
8:          $C^{HDRF}(v_1, v_2, i) = C_{REP}^{HDRF}(v_1, v_2, i) + C_{BAL}^{HDRF}(i)$
        end for
9:      for all partitions $i$ = 1 to $k$ do
10:         $ind = arg\,max_i\{C^{HDRF}(v_1, v_2, i)\}$
        end for
11:     Return $ind$     ▷ returning id of the partition
12: procedure $g(v, i)$
13:     if partition $i \in S(v)$ then
14:         Return $1 + (1 - \theta(v))$
15:     else
16:         Return 0
        end if else
```

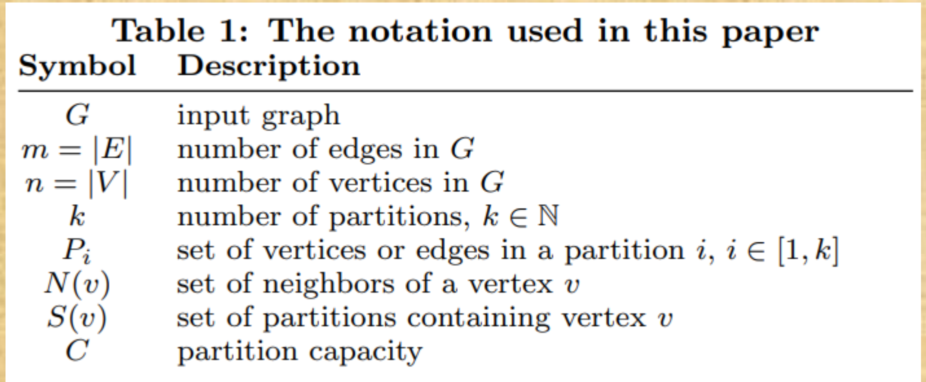


Table 1: The notation used in this paper

Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity

HDRF targets workloads with highly skewed graphs. The key idea that since power-law graphs have few high degree nodes and many low degree nodes, it is beneficial to prioritize cutting the high degree nodes to radically reduce the number of vertex-cuts. In more detail, for an input edge e = (v1, v2), the partial degrees of its endpoint vertices are recorded as δ1 and δ2 .

maxsize\minsize :the size of partition with maximum\ minimum load

S(v) :set of partitions containing vertex v

ϵ: a constant value

λ :controls load imbalance

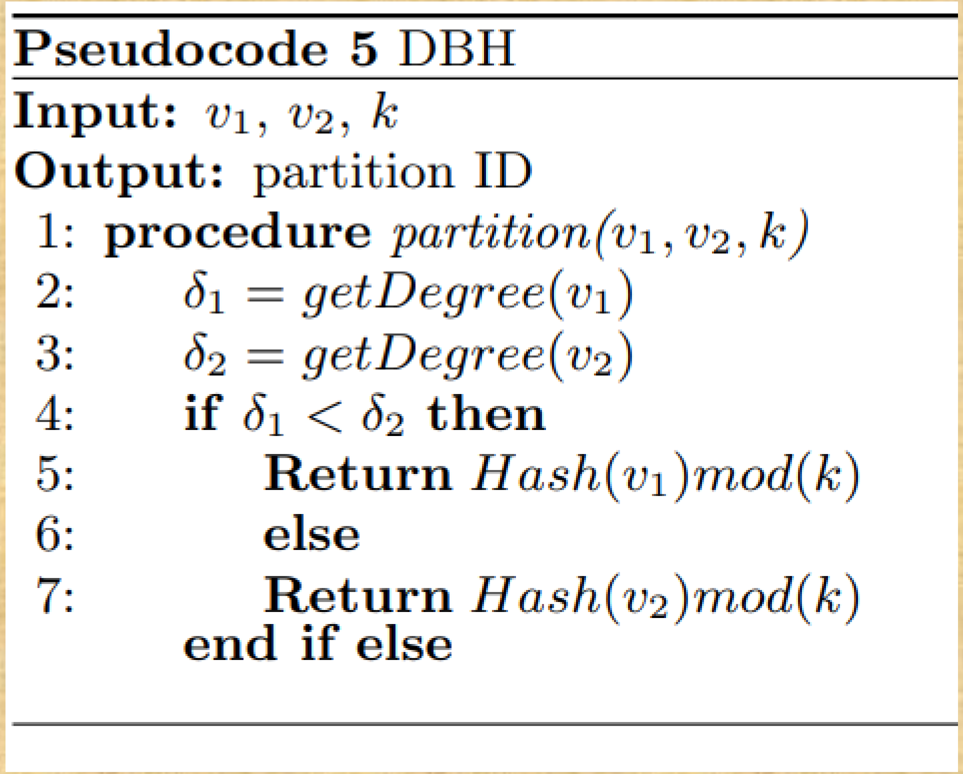
When λ ⩽ 1, the algorithm behaves similarly to Greedy.

Setting λ > 1 solves this issue by accommodating for load balance.

If λ approaches ∞, then the algorithm behaves like random hash partitioning.

Similar to Greedy, HDRF does not require any graph parameters to be computed before partitioning, and thus, it can potentially process an unbounded edge stream. On the other hand, it requires maintaining the state of partitions for computing the load and S(v).

Edge Partitioning Methods



Pseudocode 5 DBH

Input: v_1, v_2, k

Output: partition ID

```
1: procedure $partition(v_1, v_2, k)$
2:     $\delta_1 = getDegree(v_1)$
3:     $\delta_2 = getDegree(v_2)$
4:     if $\delta_1 < \delta_2$ then
5:         Return $Hash(v_1)mod(k)$
6:         else
7:         Return $Hash(v_2)mod(k)$
       end if else
```

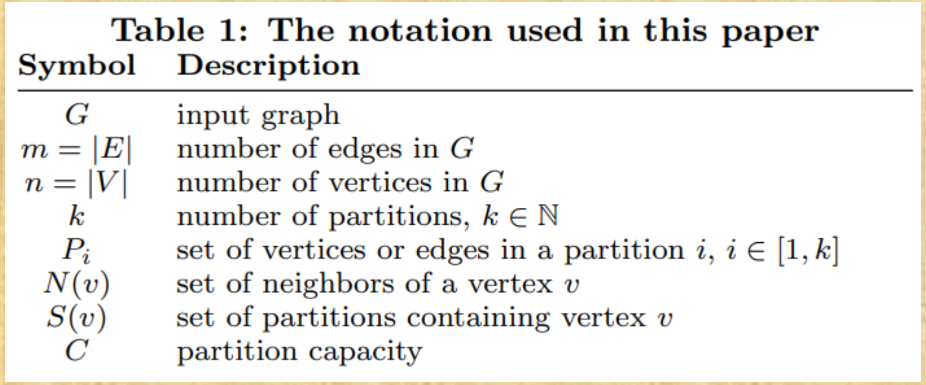


Table 1: The notation used in this paper

Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity

Degree Based Hashing is quite similar to HDRF, because it prioritizes cutting those vertices that have the lowest degree. However, unlike HDRF, DBH employs hashing for partitioning.

For an input edge e, DBH computes the partial degree of its endpoint vertices v1 and v2, δ1 and δ2. After that, e is assigned to the partition ID computed as the hash of the vertex with the lowest degree.

DBH algorithm keeps partial degree information of vertices as a state synopsis. Since it uses hashing, it can compute the current partitioning assignment on-thefly, thus reducing the state requirements. DBH can potentially process unbounded streams because no global graph properties are required prior to partitioning.

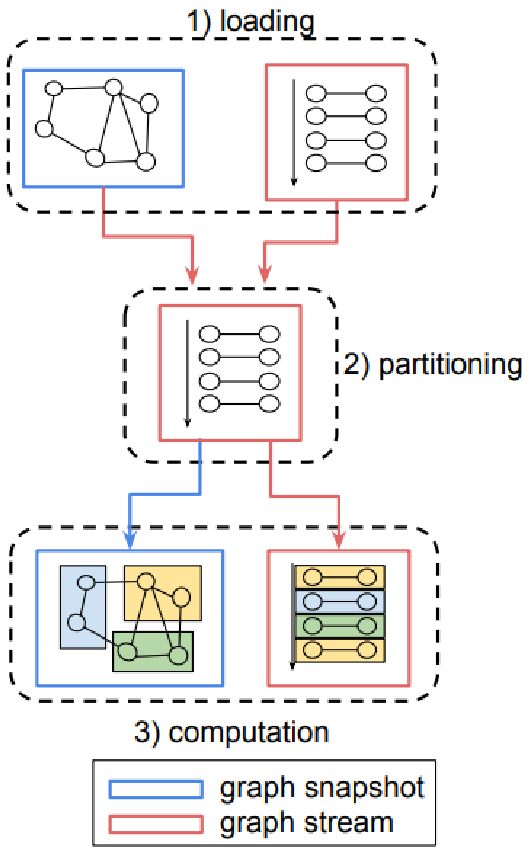


Figure 2: A general workflow for graph snapshot and stream loading, partitioning, and computation.

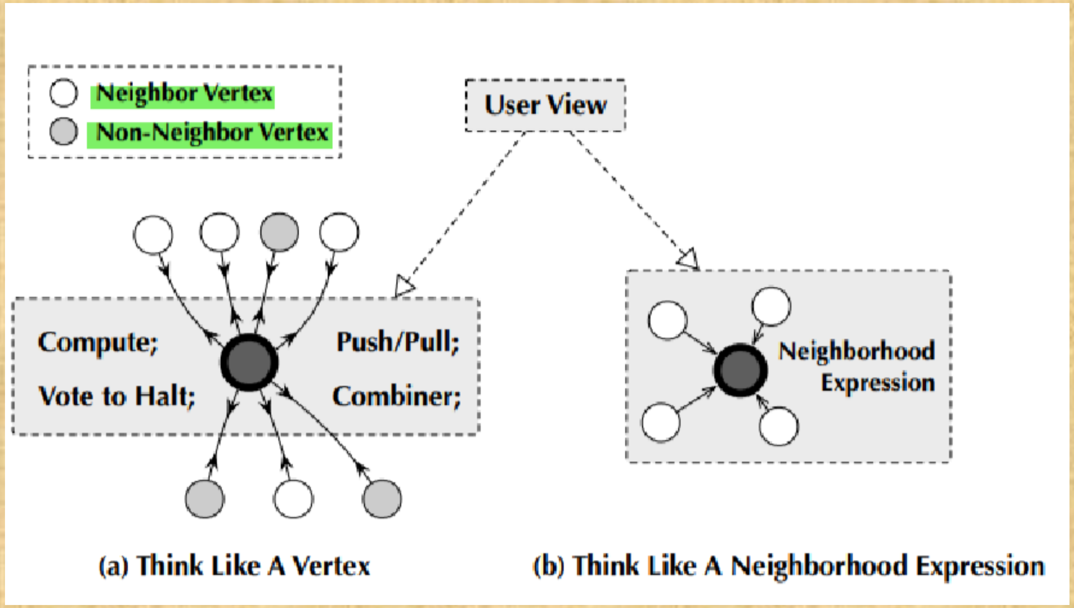
***Loading*.**
In batch processing, the graph data is bounded and represents a graph snapshot (e.g. the Facebook social network at a given time).
In stream processing, graph data is continuously read and be potentially unbounded (e.g. live user interactions on Twitter). A graph stream can be represented either as a sequence of edges (edge stream) or as a sequence of vertices with their adjacency lists (vertex stream).

***Partitioning*.**
The partitioner takes a graph stream as input and assigns each vertex or edge to a partition. In many cases loading and partitioning happen in a single phase.

***Computation*.**
In the batch model, computation starts after the whole graph has been loaded and partitioned and it operates in one or multiple passes .
In the streaming model, graph elements are only accessed once. Therefore, applications that employ on the-fly processing are also referred as single-pass streaming applications.

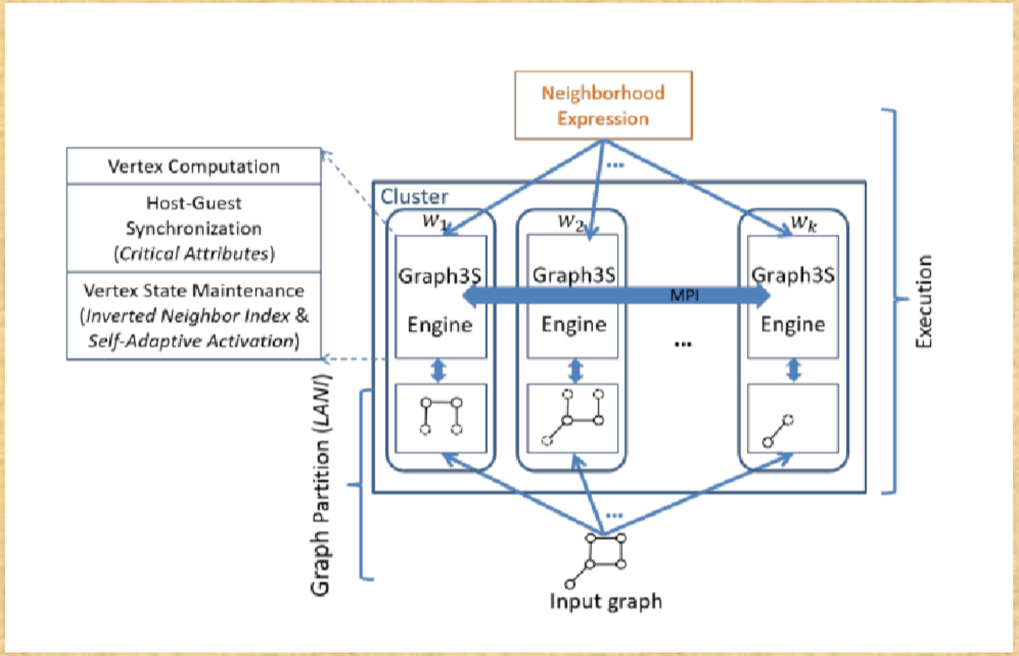
Flexibility-Simplicity

Simplicity means easy to use, easy programming interface.
An easy programming interface should receive as much as small information from user side when complete the computation of a specific problem.



Flexibility-Efficiency

- High efficiency is critical for graph processing systems.
- Efficiency should simplify the workload on the user side, the computation workload, communication overhead, vertex state maintenance, and optimization realizations.



1. Different from pushing/pulling required multiple vertex attributes in existing systems, only changed vertex attributes (critical attributes), are synchronized in Graph3S which reduces communication cost.
2. A dual neighbor index is designed to accelerate vertex computation and activation procedure.
3. A self-adaptive activation mechanism is proposed further improves the efficiency of Graph3S.

Flexibility-Scalability

Scalability for distributed graph processing systems is important for the large-scale processing.

One technique to improve scalability is to store the vertex information & neighborhood information into the memory. There are three reasons for this:

1. Vertex info O(V), neighborhood info O(V^2)
2. Edge info is rarely changed, vertex more frequently changed
3. Vertex only communicate with neighborhood, make use of memory

Balance problem in graph partition

Balance problem is an important part in graph partition.
Balance problem has into two parts.

Part Ⅰ: load balance.
We need to evenly distributed the large scale graph in case that single machine can’t store the whole graph.(evenly here means parts in different machines basically take same space)

Part Ⅱ: workload balance.
We need to evenly distributed the vertices for balanced workload while minimizing inter-partition edges to avoid costly network traffic.(evenly here means the workload of machines should basically same as each other, but the load balance may have differences)

The graphs resource we get can be roughly divided into two kinds, one is static and another is dynamic.

Static kinds take a whole graph as input. This kind of graph usually has relative small scale. We can achieve load balance while workload balance at the same time if the graph is not that massive.

Dynamic kinds take vertices stream or edges stream as input, the load problem always happens when receiving this kind of input.

Part Ⅰ: Load balance

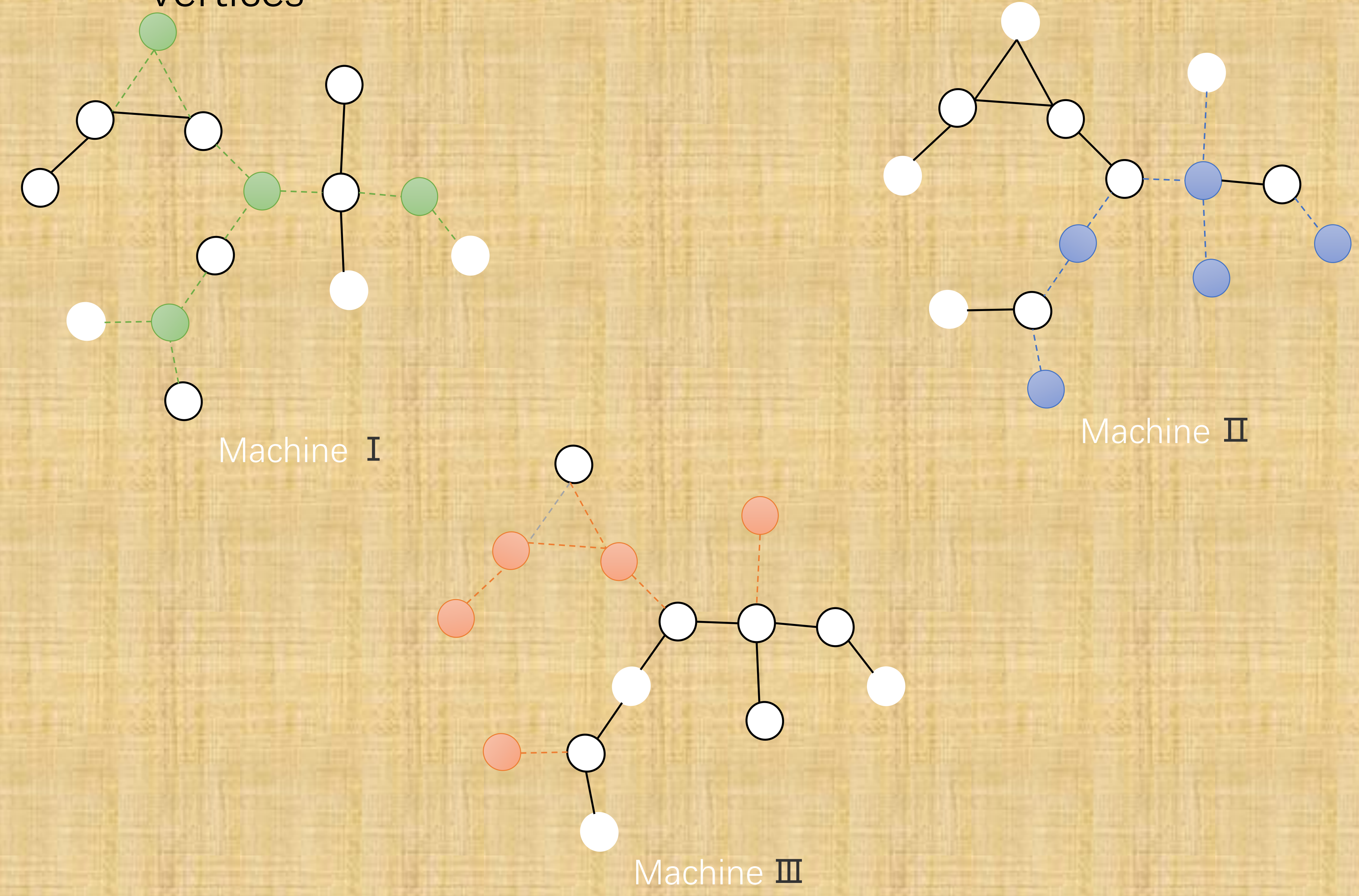
Load balance happens in the very beginning, this part is about to evenly distributed a large graph.

For example, in social media graphs usually have trillion edges, so they can’t be directly stored into a single machine restricted to the memory size. In this case, we need to evenly distributed them into serval machines.

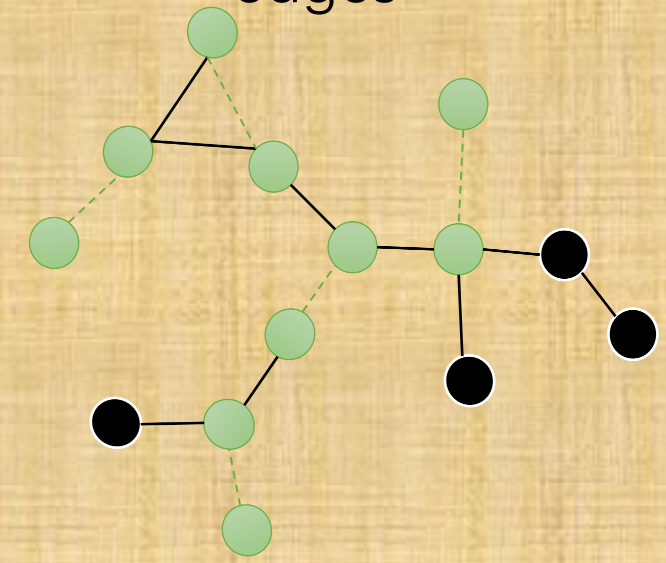
This distribution is always random, hash method is a most common method.

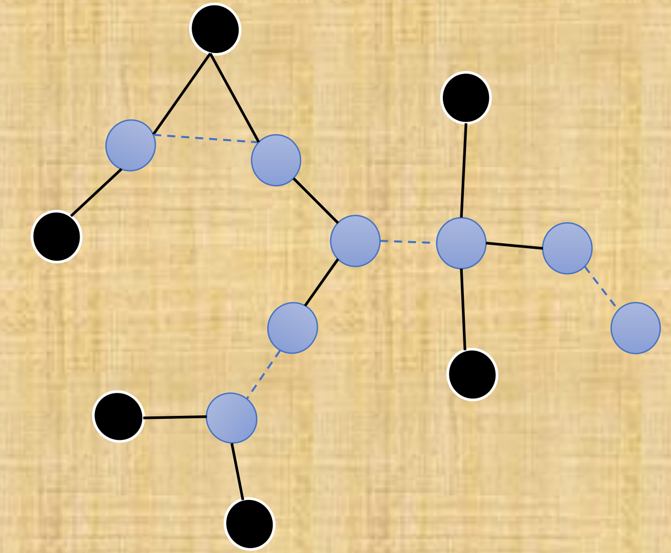
There are two ways to store a graph, one’s benchmark against vertices and another’s benchmark against edges.

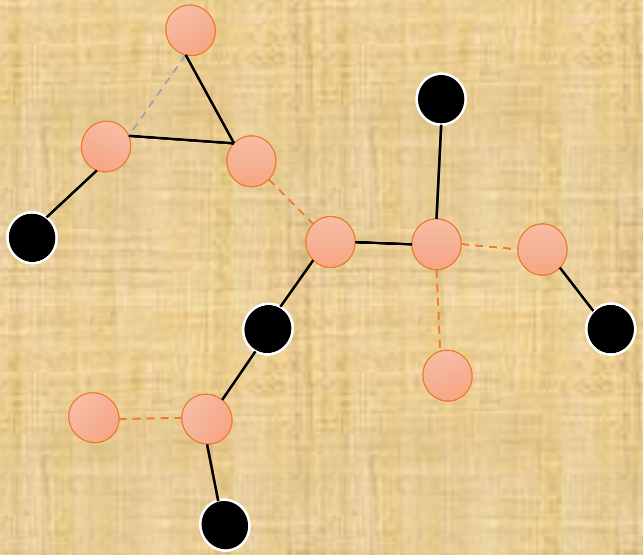
Example for randomly storing a graph based on vertices



Example for randomly storing a graph based on edges







Part Ⅱ: Workload balance

Workload balance is the most important part of balanced distributed a graph.

Workload balance means when running distributed algorithm on the cluster, every machine should finish their task basically in a same period of time. This is always achieved by a great partition.

Different from load balance problem, workload balance need to take more factors into consideration. Not only the number of vertices and edges in every partition be considered, but the problem we need to solve and the topology of the graph also need to be covered.

Every different problems on a same graph may need different partitions to achieve workload balance.

However, topology of a graph is usually hard to be transformed into computable form. So most of the algorithms only take the number of vertices or edges in every partition as the judgement of workload balance.

In this case, workload balance is equal to load balance, so the realization of them happens at the same time.

Workload balance is realized by different partition algorithms.

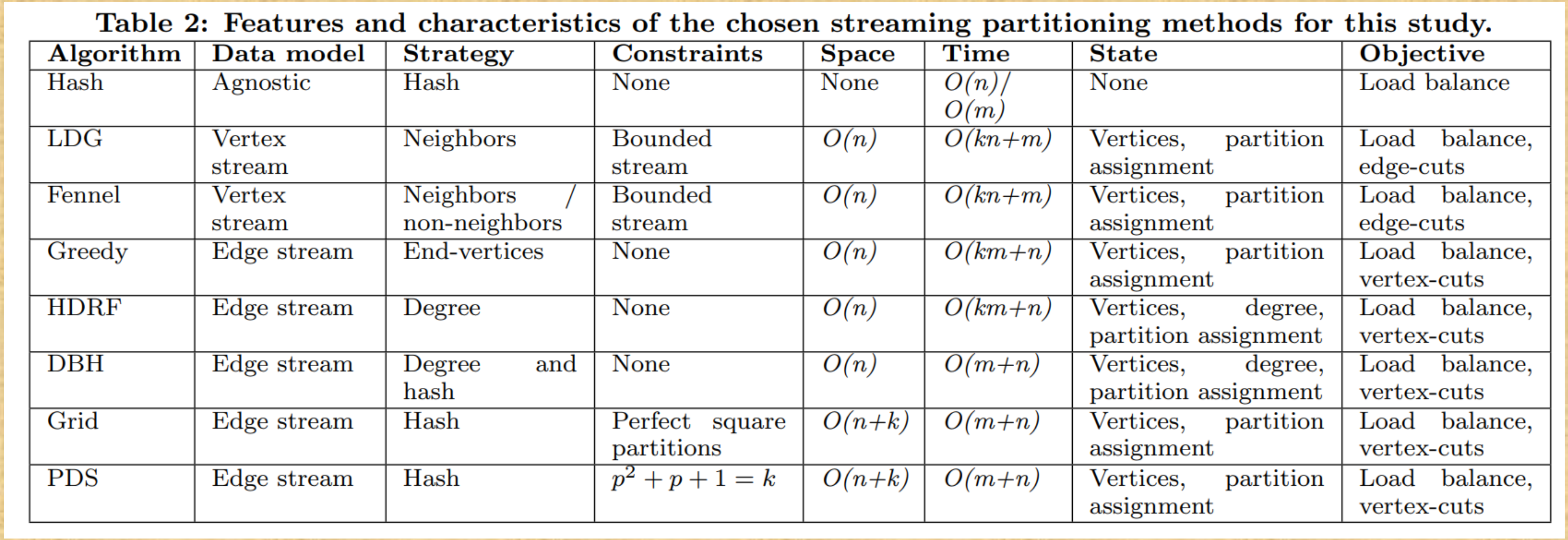


Table 2: Features and characteristics of the chosen streaming partitioning methods for this study.

Algorithm	Data model	Strategy	Constraints	Space	Time	State	Objective
Hash	Agnostic	Hash	None	None	$O(n)$/ $O(m)$	None	Load balance
LDG	Vertex stream	Neighbors	Bounded stream	$O(n)$	$O(kn+m)$	Vertices, partition assignment	Load balance, edge-cuts
Fennel	Vertex stream	Neighbors / non-neighbors	Bounded stream	$O(n)$	$O(kn+m)$	Vertices, partition assignment	Load balance, edge-cuts
Greedy	Edge stream	End-vertices	None	$O(n)$	$O(km+n)$	Vertices, partition assignment	Load balance, vertex-cuts
HDRF	Edge stream	Degree	None	$O(n)$	$O(km+n)$	Vertices, degree, partition assignment	Load balance, vertex-cuts
DBH	Edge stream	Degree and hash	None	$O(n)$	$O(m+n)$	Vertices, degree, partition assignment	Load balance, vertex-cuts
Grid	Edge stream	Hash	Perfect square partitions	$O(n+k)$	$O(m+n)$	Vertices, partition assignment	Load balance, vertex-cuts
PDS	Edge stream	Hash	$p^2+p+1=k$	$O(n+k)$	$O(m+n)$	Vertices, partition assignment	Load balance, vertex-cuts

Task Scheduling Framework-Fork Join

- Input vertices are partitioned using a Scheduling Strategy resulting in a list of (worker, partition) pairs, as shown in (1).
- A user-defined Task is created for each (worker, partition) pair, as shown in (2).
- Each {K0, K1,…,KR} emitted by the workers represent the intermediate results and have an associated value (3).
- As Tasks complete execution and return results, the join function merges sub-results (4).

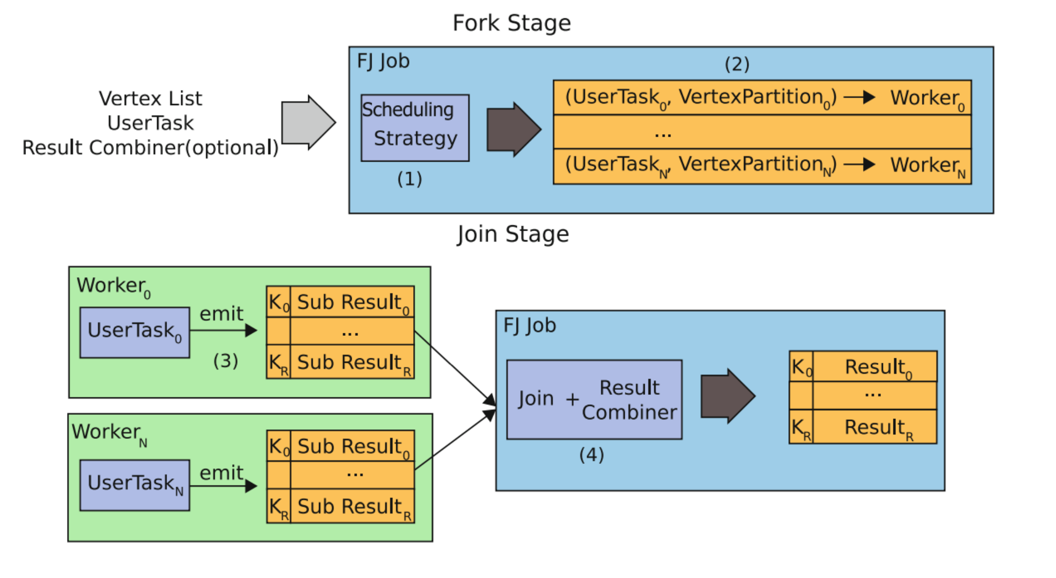


Fig. 1. The Fork-Join workflow in graphly.

Task Scheduling Framework-Pregel

1. A Pregel Client
2. A coordinator
3. A Pregel Worker

1. Pregel Client receives user input.
2. Coordinator sends activation messages to all vertices, splits the graph into partitions then sends them to the workers, makes local state aggregators.
3. PregelWorkers receives the information.
4. Computation locally on workers with active state vertices.
5. all messages generated by the execution are sent to their corresponding workers.
6. A combiner is applied to merge messages from different workers to reduce memory consumption.

Different from Fork-Join, the Pregel scheduling mechanism uses a worker2worker messaging mechanism instead of a global one.

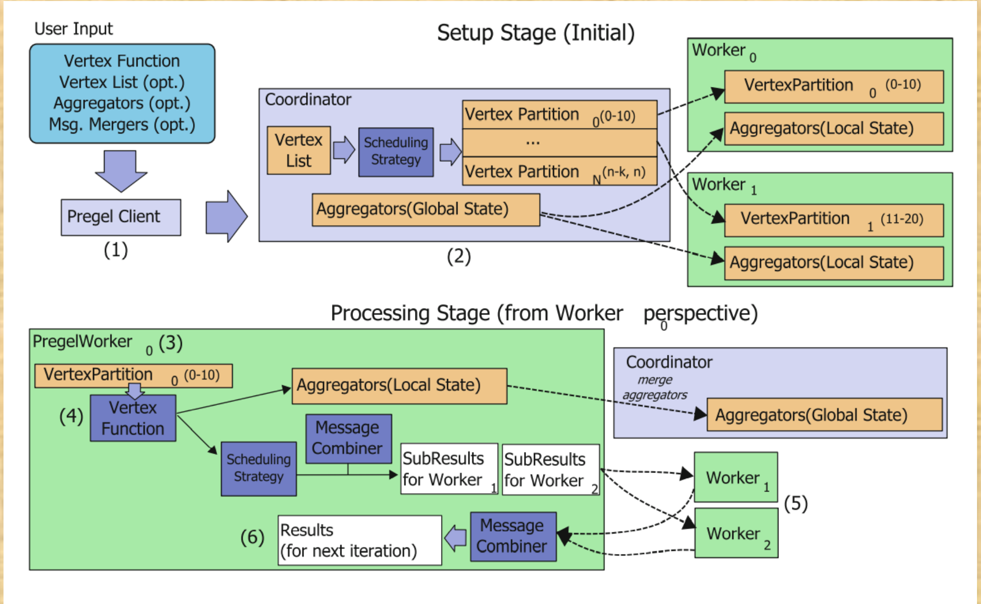


Fig. 2. The Pregel workflow in graphly.

Task Scheduling Framework-DPM

While centralization can reduce the communication strategy and use less computation resources, it is harmful to the distributed computing.
In DPM, it introduce SubResult worker to decentralize the computation.

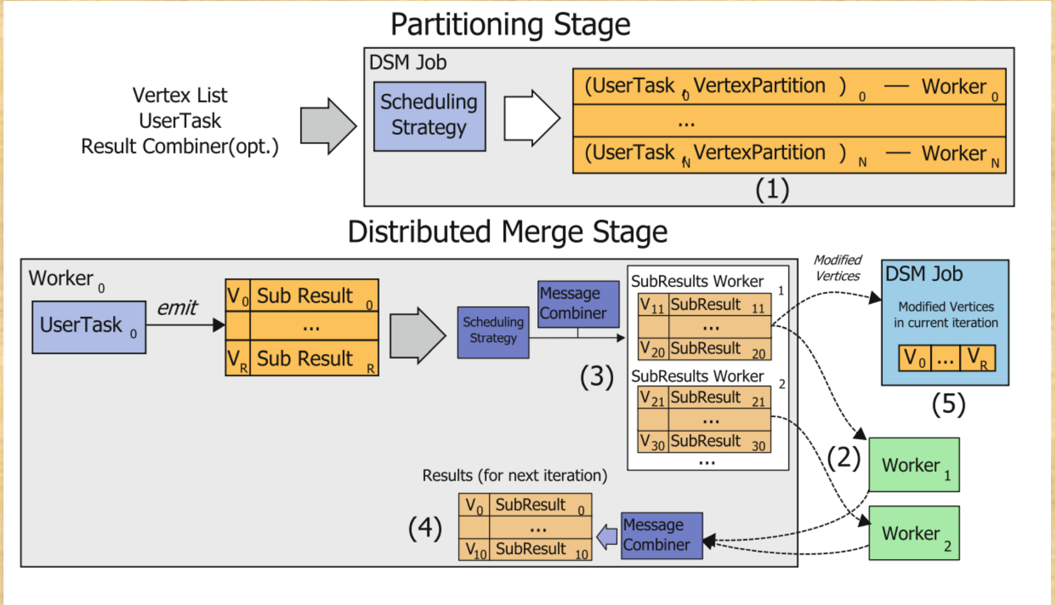


Fig. 3. Distributed partitioned merge model: Overview

Task Scheduling Methods

- Location aware
  - Each vertex has its specific worker to process it.
- Round Robin
  - Evenly distribute the vertices of computation request to the available workers.
- Available Memory
  - Obtains the memory currently available on each worker and then divides the given list of vertices accordingly. (Dynamic Methods, Immediate Computation)
- Maximum Memory
  - Considers the maximum amount of memory that a worker can use to divide the list of vertices. (Static Methods, Delay Computation)

Fault Tolerance

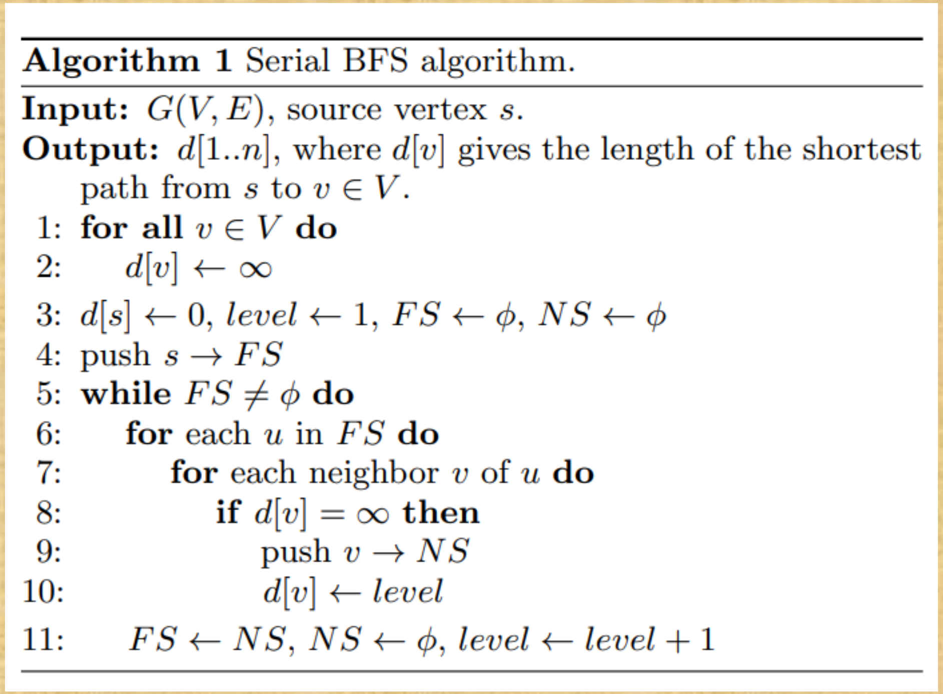
- Breakpoint Recovery
  - Recover to the breakpoint before the fault occur.
- Logging
  - Logging the non-deterministic events in a state interval form to recover them when needed.
- Replication
  - Make multiple replicas for each data block.
- Algorithm Compensation
  - Using algorithm to generate the missing state.

Algorithm Implementation

- Distributed Graph Algorithms Processing
    - BFS
    - Dijkstra

Distributed-BFS-Paper: Parallel Breadth-First Search on Distributed Memory Systems

- Serial BFS:



Algorithm 1 Serial BFS algorithm.

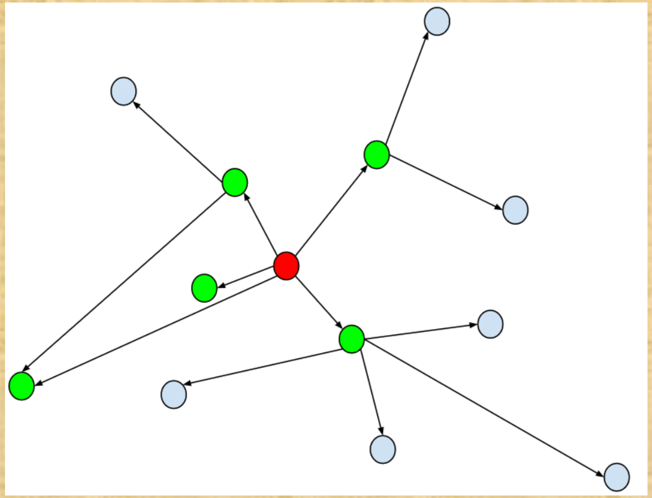
```
Input: $G(V, E)$, source vertex $s$.
Output: $d[1..n]$, where $d[v]$ gives the length of the shortest
    path from $s$ to $v \in V$.
 1: for all $v \in V$ do
 2:     $d[v] \leftarrow \infty$
 3: $d[s] \leftarrow 0$, $level \leftarrow 1$, $FS \leftarrow \phi$, $NS \leftarrow \phi$
 4: push $s \rightarrow FS$
 5: while $FS \neq \phi$ do
 6:     for each $u$ in $FS$ do
 7:         for each neighbor $v$ of $u$ do
 8:             if $d[v] = \infty$ then
 9:                 push $v \rightarrow NS$
10:                 $d[v] \leftarrow level$
11:     $FS \leftarrow NS$, $NS \leftarrow \phi$, $level \leftarrow level + 1$
```

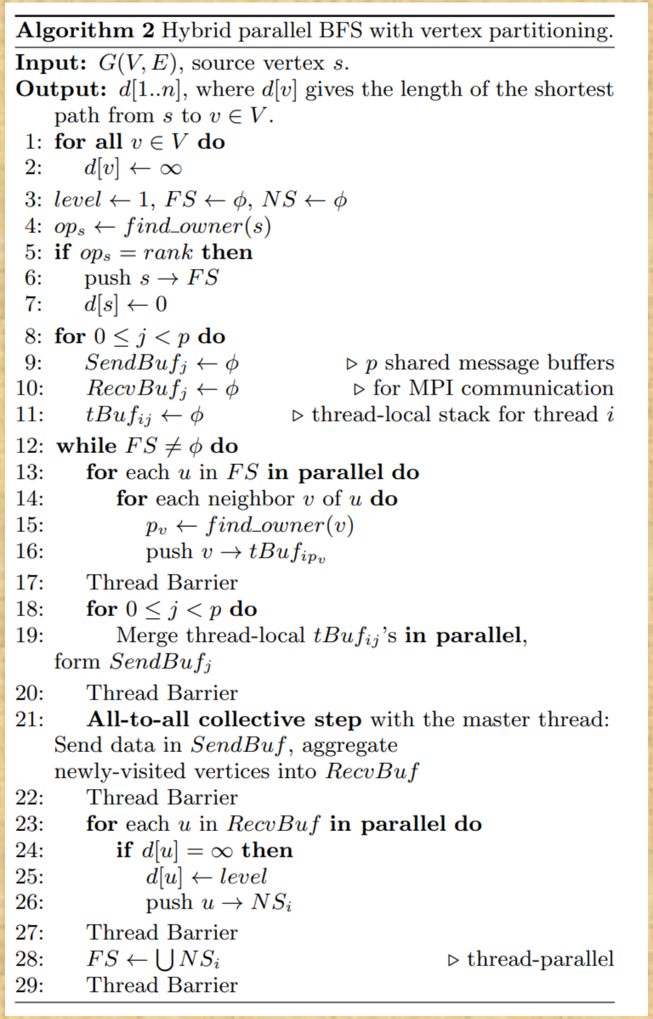
Breadth-first ordering of vertices is accomplished in this case by using two stacks – FS and NS – for storing vertices at the current level (or “frontier”) and the newly-visited set of vertices (one hop away from the current level) respectively. The number of iterations of the outer while loop (lines 5- 11) is bounded by the length of the longest shortest path from s to any reachable vertex t. Note that this algorithm is slightly different from the widely-used queue-based serial algorithm. We can relax the FIFO ordering mandated by a queue at the cost of additional space utilization, but the work complexity in the RAM model is still O(m + n).

Distributed-BFS-Paper: Parallel Breadth-First Search on Distributed Memory Systems

- BFS with 1D Partitioning

One-dimensional decomposition of the adjacency matrix corresponding to the graph.
The general schematic of the level-synchronous parallel BFS algorithm can be modified to work in a distributed scenario with 1D partitioning as well.
The distance array is distributed among processes. Every process only maintains the status of vertices it owns, and so the traversal loop becomes an edge aggregation phase.



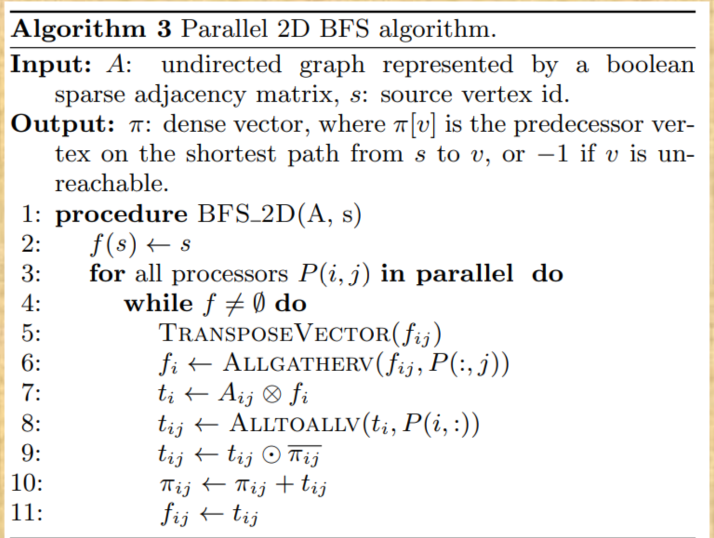


Algorithm 2 Hybrid parallel BFS with vertex partitioning.

```
Input: $G(V, E)$, source vertex $s$.
Output: $d[1..n]$, where $d[v]$ gives the length of the shortest
    path from $s$ to $v \in V$.
 1: for all $v \in V$ do
 2:     $d[v] \leftarrow \infty$
 3: $level \leftarrow 1$, $FS \leftarrow \phi$, $NS \leftarrow \phi$
 4: $op_s \leftarrow find\_owner(s)$
 5: if $op_s = rank$ then
 6:     push $s \rightarrow FS$
 7:     $d[s] \leftarrow 0$
 8: for $0 \leq j < p$ do
 9:     $SendBuf_j \leftarrow \phi$          ▷ $p$ shared message buffers
10:     $RecvBuf_j \leftarrow \phi$          ▷ for MPI communication
11:     $tBuf_{ij} \leftarrow \phi$          ▷ thread-local stack for thread $i$
12: while $FS \neq \phi$ do
13:     for each $u$ in $FS$ in parallel do
14:         for each neighbor $v$ of $u$ do
15:             $p_v \leftarrow find\_owner(v)$
16:             push $v \rightarrow tBuf_{ip_v}$
17:     Thread Barrier
18:     for $0 \leq j < p$ do
19:         Merge thread-local $tBuf_{ij}$'s in parallel,
    form $SendBuf_j$
20:     Thread Barrier
21:     All-to-all collective step with the master thread:
    Send data in $SendBuf$, aggregate
    newly-visited vertices into $RecvBuf$
22:     Thread Barrier
23:     for each $u$ in $RecvBuf$ in parallel do
24:         if $d[u] = \infty$ then
25:             $d[u] \leftarrow level$
26:             push $u \rightarrow NS_i$
27:     Thread Barrier
28:     $FS \leftarrow \bigcup NS_i$          ▷ thread-parallel
29:     Thread Barrier
```

Distributed-BFS-Paper: Parallel Breadth-First Search on Distributed Memory Systems

In 2D decomposition, vertex ownerships are more flexible. One practice is to distribute the vector entries only over one processor dimension (pr or pc), for instance the diagonal processors if using a square grid (pr = pc), or the first processors of each processor row. This approach is mostly adequate for sparse matrix-dense vector multiplication (SpMV).



Algorithm 3 Parallel 2D BFS algorithm.

Input: A: undirected graph represented by a boolean sparse adjacency matrix, s: source vertex id.

Output: π: dense vector, where $\pi[v]$ is the predecessor vertex on the shortest path from s to v, or -1 if v is unreachable.

```
procedure BFS_2D(A, s)
    $f(s) \leftarrow s$
    for all processors $P(i,j)$ in parallel do
        while $f \neq \emptyset$ do
            TransposeVector($f_{ij}$)
            $f_i \leftarrow$ Allgatherv($f_{ij}, P(:,j)$)
            $t_i \leftarrow A_{ij} \otimes f_i$
            $t_{ij} \leftarrow$ Alltoallv($t_i, P(i,:)$)
            $t_{ij} \leftarrow t_{ij} \odot \overline{\pi_{ij}}$
            $\pi_{ij} \leftarrow \pi_{ij} + t_{ij}$
            $f_{ij} \leftarrow t_{ij}$
```

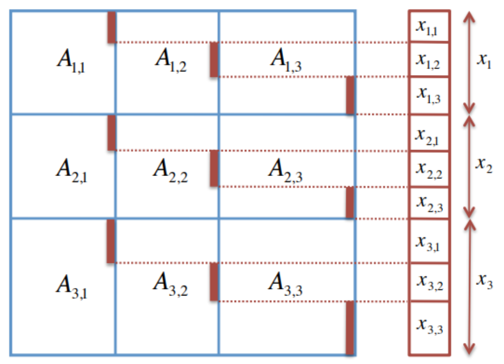
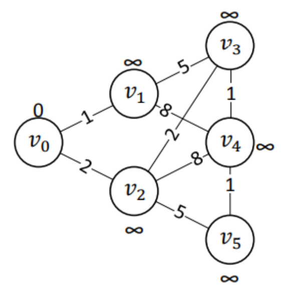


Figure 1: 2D vector distribution illustrated by interleaving it with the matrix distribution.

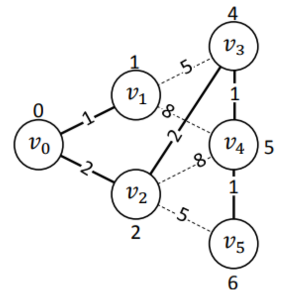
Distributed-BFS-Paper: Parallel Breadth-First Search on Distributed Memory Systems

- Shared Memory Computation
  - pushes of newly-visited vertices to the stack NS
  - consider the distance checks and updates
- Distributed-memory parallelism
  - MPI message-passing library to express the inter-node communication steps. Utilize the collective communication routines Alltoallv, Allreduce, and Allgather.
- Load-balancing traversal
  - Randomly shuffling all the vertex identifiers prior to partitioning, which leads to each process getting roughly the same number of vertices and edges, regardless of the degree distribution.

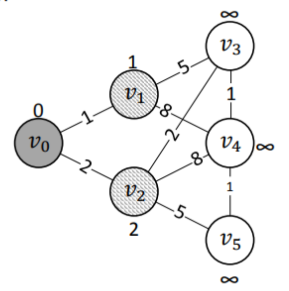
Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem



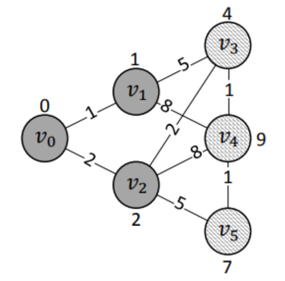
(a) Initial setup. $d(v_1) = \cdots = d(v_5) = \infty$ and $d(v_0) = 0$.



(b) Solution. The bold solid lines show the shortest paths.



(c) Vertex v_0 is relaxed and thus, vertices v_1 and v_2 are activated.



(d) Vertices v_0 and v_1 are relaxed and vertices v_2 and v_3 are active.

Figure 1: An instance of SSSP problem with v_0 as the source vertex. The values next to vertices are the current distances of vertices.

Dijkstra:

This algorithm relaxes active vertices in current distance order, relax the active vertex with the minimum current distance at each iteration.

Bellman-Ford & Chaotic Relaxation Algorithm:

The Bellman-Ford's algorithm relaxes all vertices |V (G)| (number of vertices) times. Chaotic Relaxation is Bellman-Ford's algorithm with a condition that each time only relaxes the active vertices.

Δ-Stepping:

In Δ-Stepping, i iterates increasingly in {0, 1, 2, . . . }. For each i, the set of active vertices that can be relaxed are those vertices v that i.Δ ⩽ d(v) < (i + 1).Δ where Δ is a constant throughout the algorithm. Δ-Stepping provides two benefits: performing a close-to minimum amount of work while having a reasonable amount of parallelism. Note that Δ-Stepping with Δ = 1 is equivalent to parallel Dijkstra's (assume that edge weights are integers) and with Δ = ∞ is equivalent to Chaotic Relaxation.

Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

1. Distributor: breaks the graph into P (total number of processors) subgraphs so that all subgraphs have approximately the same number of edges. Each subgraph is assigned to a different processor. The owner of each vertex computes the shortest distance to that vertex from the source.
2. Pruning: detects edges that can be guaranteed not to be used for any shortest path from any source vertex.
3. Subgraph Extraction: extracts a subgraph of the input such that the shortest paths go through that subgraph.
4. DSMR: computes all the shortest paths from a given source vertex.
5. Fix-up: fix the potential incorrect computation caused by the ignorance of part of the graphs by the subgraph extraction.

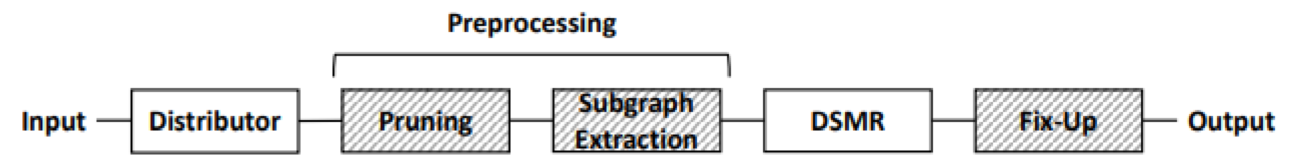


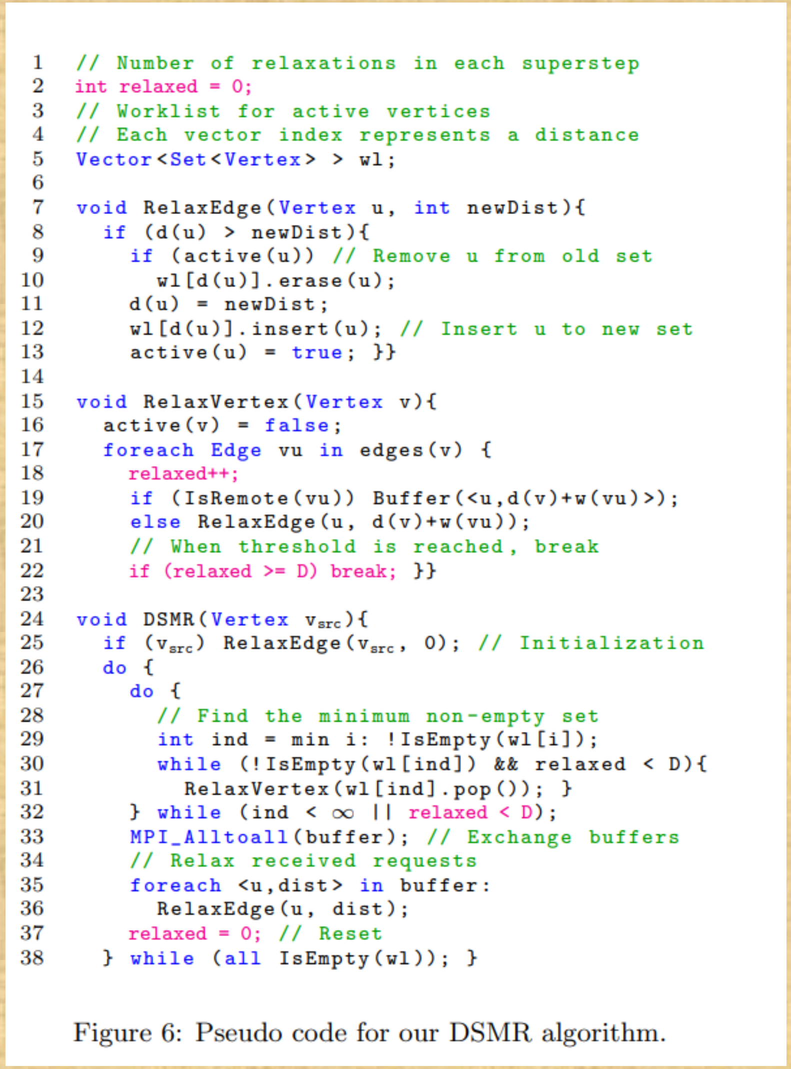
Figure 2: Overview of the engines in our algorithm.

Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

- The Distributor
- There are two major concerns in data distribution of a scale-free graph on a distributed-memory system: **existence of high-degree vertices** and **vertices assignment** to processors.
- Scale-free networks have a few high-degree vertices and many low degree vertices. Assigning a high-degree vertex to a single processor increases the likelihood of load imbalance, which achieved by Vertex Splitting.
- The Distributor engine accepts as an input a threshold and the vertices with degree higher than that are copied on each of the P processors and 1 P th of edges of the original vertex are assigned to each of the processors. These P copies are connected to a unique copy by P edges with weight 0 which guarantees equal shortest distances for all P + 1 copies. The low degree vertices are shuffled using a random permutation. Then, they are assigned to processors in consecutive chunks such that the number of edges for each processor is roughly the same. The output of the distributor will be P subgraphs with disjoint vertex sets. However, the edges joining these subgraphs are shared by the processors containing the subgraphs. Each subgraph has an equal number of edges and, consequently, the number of vertices are not necessarily equal. The owner of each vertex is responsible for computing the shortest distance to that vertex.

Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

- DSMR



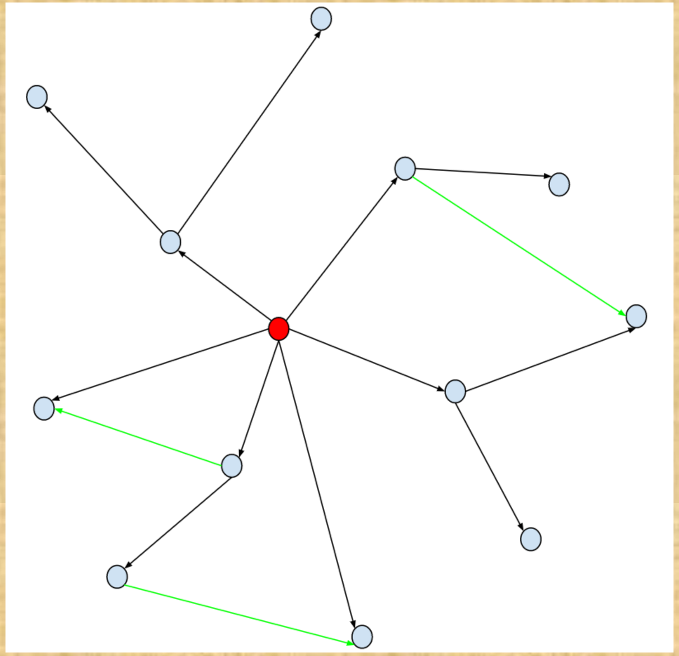
```
// Number of relaxations in each superstep
int relaxed = 0;
// Worklist for active vertices
// Each vector index represents a distance
Vector<Set<Vertex> > wl;

void RelaxEdge(Vertex u, int newDist){
  if (d(u) > newDist){
    if (active(u)) // Remove u from old set
      wl[d(u)].erase(u);
    d(u) = newDist;
    wl[d(u)].insert(u); // Insert u to new set
    active(u) = true; }}

void RelaxVertex(Vertex v){
  active(v) = false;
  foreach Edge vu in edges(v) {
    relaxed++;
    if (IsRemote(vu)) Buffer(<u,d(v)+w(vu)>);
    else RelaxEdge(u, d(v)+w(vu));
    // When threshold is reached, break
    if (relaxed >= D) break; }}

void DSMR(Vertex $v_{src}$){
  if ($v_{src}$) RelaxEdge($v_{src}$, 0); // Initialization
  do {
    do {
      // Find the minimum non-empty set
      int ind = min i: !IsEmpty(wl[i]);
      while (!IsEmpty(wl[ind]) && relaxed < D){
        RelaxVertex(wl[ind].pop()); }
    } while (ind < $\infty$ || relaxed < D);
    MPI_Alltoall(buffer); // Exchange buffers
    // Relax received requests
    foreach <u,dist> in buffer:
      RelaxEdge(u, dist);
    relaxed = 0; // Reset
  } while (all IsEmpty(wl)); }
```

Figure 6: Pseudo code for our DSMR algorithm.



Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

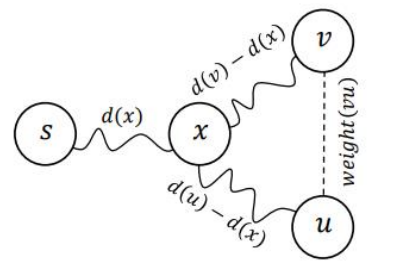
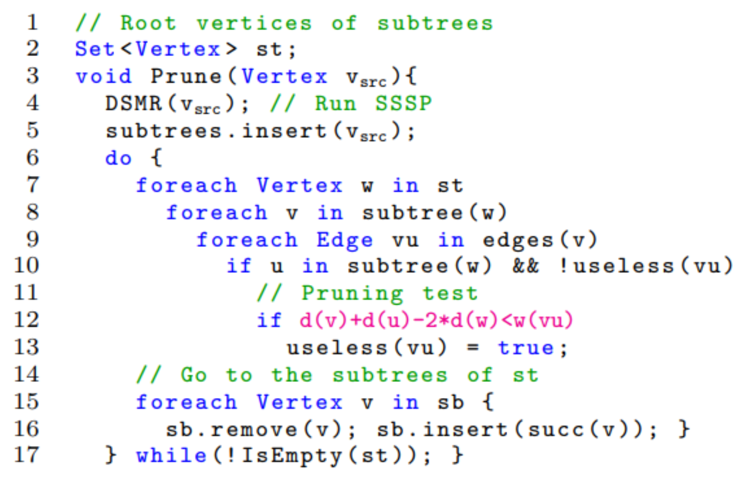


Figure 9: Pruning idea. x is the first common ancestor of v and u.



```
// Root vertices of subtrees
Set<Vertex> st;
void Prune(Vertex v_src){
  DSMR(v_src); // Run SSSP
  subtrees.insert(v_src);
  do {
    foreach Vertex w in st
      foreach v in subtree(w)
        foreach Edge vu in edges(v)
          if u in subtree(w) && !useless(vu)
            // Pruning test
            if d(v)+d(u)-2*d(w)<w(vu)
              useless(vu) = true;
    // Go to the subtrees of st
    foreach Vertex v in sb {
      sb.remove(v); sb.insert(succ(v)); }
  } while(!IsEmpty(st)); }
```

Figure 10: Pseudo code for Prune engine.

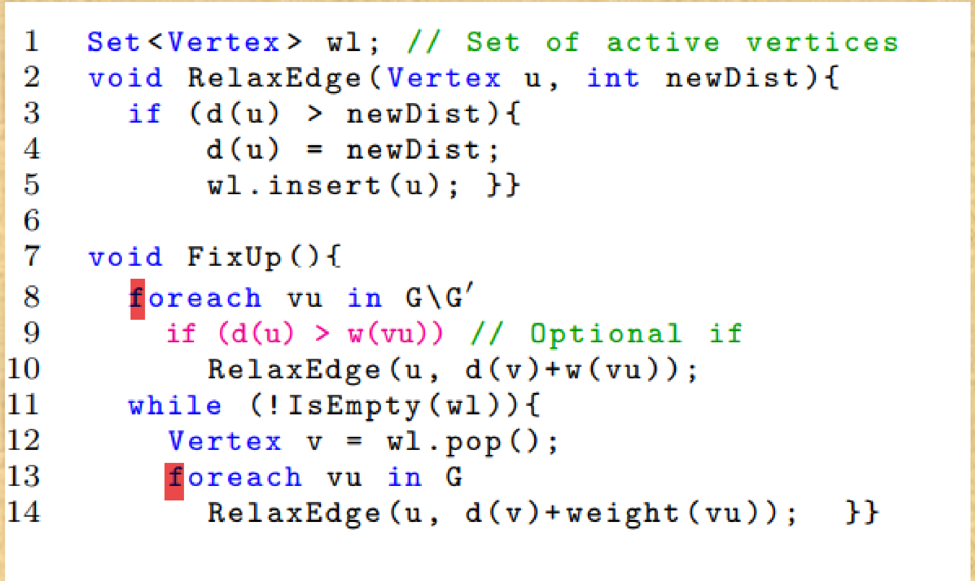
- Pruning
- Pruning is a preprocessing technique that identifies edges in a graph G which can be guaranteed not to be used in any shortest path from any source vertex. The left figure shows a pruning scenario where edge vu, shown as a dotted line, is a candidate for pruning. Our pruning engine chooses a random source vertex s (not necessarily the one used by the DSMR engine) and computes the shortest distances for all vertices. In the figure it shows the shortest paths from the chosen source vertex s to u and v by solid wavy lines. Assume that these two shortest path diverge at vertex x. We call x the first common ancestor of v and u. If the condition (d(u) − d(x)) + (d(v) − d(x)) < w(vu) is true, vu is marked useless. We call this the pruning test. The reason why this edge can be marked as useless and ignored when computing the shortest path from any source vertex is that the paths from x to v and from x to u are of distance d(v) − d(x) and d(u)−d(x), respectively. Therefore, if the distances of these two paths together are less than w(vu), the edge vu will not be used in any shortest path since when going from u to v (or from v to u) is always shorter to go through x.

Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

- Graph Extraction
- Graph extraction is a preprocessing technique that extracts a subgraph $G_0 \subseteq G$ from the input graph G, such that most of the shortest paths in G go through G_0.
- Once G_0 is computed, the shortest paths are computed in two phases:
  - First, DSMR is executed with G_0 to compute the shortest distances in G_0. After this is done, the shortest distances for most vertices of G would have been computed correctly.
  - Then, for the rest of the graph, $G \setminus G_0$, the Fix-Up engine corrects the distances of the vertices in G computed incorrectly in the first phase. Using the edges in $G \setminus G_0$ the Fix-up algorithm updates the distances of only a few vertices and consequently, relaxing these vertices in any order will not cause significant work overhead.

Distributed-SSSP-Paper: DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

- Fix-up phase implementation



```
Set<Vertex> wl; // Set of active vertices
void RelaxEdge(Vertex u, int newDist){
  if (d(u) > newDist){
      d(u) = newDist;
      wl.insert(u); }}

void FixUp(){
  foreach vu in G\G'
    if (d(u) > w(vu)) // Optional if
      RelaxEdge(u, d(v)+w(vu));
  while (!IsEmpty(wl)){
    Vertex v = wl.pop();
    foreach vu in G
      RelaxEdge(u, d(v)+weight(vu));  }}
```

Figure 8: Pseudo code for Fix-Up engine.

BFS-Distributed Graph Processing System View

Graph Partition	Edge Partition
Communication	Collective Communication, buffer the possible expansion where out-neighbor and the original FS vertex in different machines.
Task Schedule	BSP Superstep, each superstep is defined as FS->NS
Fault Tolerance	
Load Balanced	Random Partition, without considering degree distribution
Coding Technique	2D adj matrix & processor storage and index

SSSP-Distributed Graph Processing System View

Graph Partition	Edge Partition
Communication	Collective Communication, buffered the possible edge relaxation where two end vertices are stored in two different machines.
Task Schedule	BSP Superstep, each superstep is defined as reaching the maximum edge relaxation number (since the edge relaxation will always result in memory access, which is time-consuming)
Fault Tolerance	
Load Balanced	Vertices with a degree that higher than a preset threshold will be duplicated to all machines.
Coding Technique	Subgraph Extraction & Fix-up, Pruning