Domain-specific Programming on Graphs

Xuhao Chen

cxh@mit.edu

March 13, 2023

Massachusetts Institute of Technology

CSAIL

Domain-specific Programming

- A **domain-specific** programming language/system is a computer language/system specialized to a particular application domain

matrix computing

deep learning

relational database

image processing

hardware description languages

Why Domain-specific Programming?

High Productivity

MATLAB

```
a = randi([0, 1], [10,10]);

b = randi([0, 1], [10,10]);

c = a * b;
```

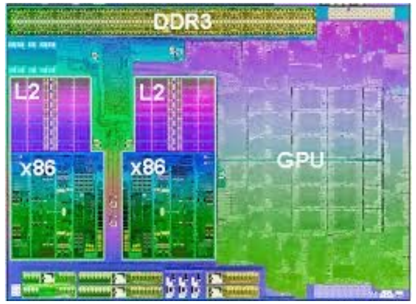
```
#include<stdio.h>
int main() {
 int a[10][10], b[10][10], c[10][10], n=10, i, j, k;
 for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
     init_rand(&a[i][j]);
     init_rand(&b[i][j]);
   }
 }
```

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int im = 0; im < s; im += t)
        for (int jm = 0; jm < s; jm += t)
          for (int km = 0; km < s; km += t)
            for (int il = 0; il < t; ++il)
              for (int kl = 0; kl < t; ++kl)
                for (int jl = 0; jl < t; ++jl)
                  C[ih+im+il][jh+jm+jl] +=
                    A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
```

Heterogeneous Parallel Platforms

Multicore CPU

Integrated CPU + GPU



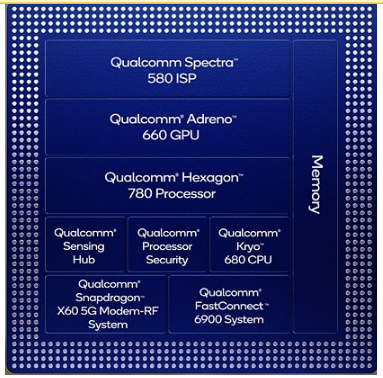
GPU

throughput cores + fixed-function

FPGA

programmable hardware

How do we enable **programmers** to **productively write software** that **efficiently** uses current and future **heterogeneous, parallel machines**?

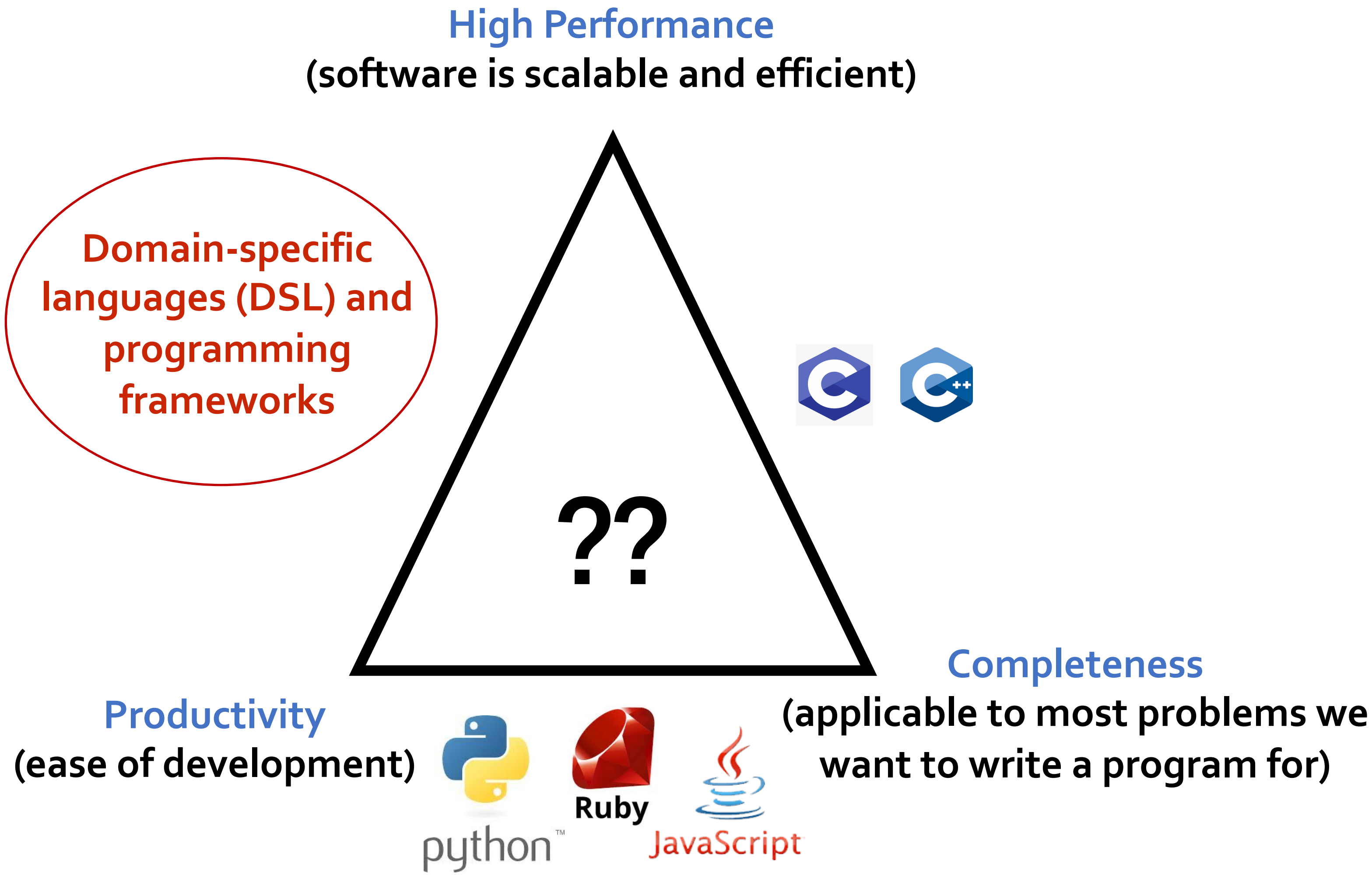


JavaScript, Swift, Renderscript

Abstractions: message passing

MPI, Go channels, Spark, Charm++

The [magical] ideal parallel programming language



Credit: Pat Hanrahan

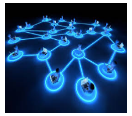
Domain-specific Programming System for Graphs

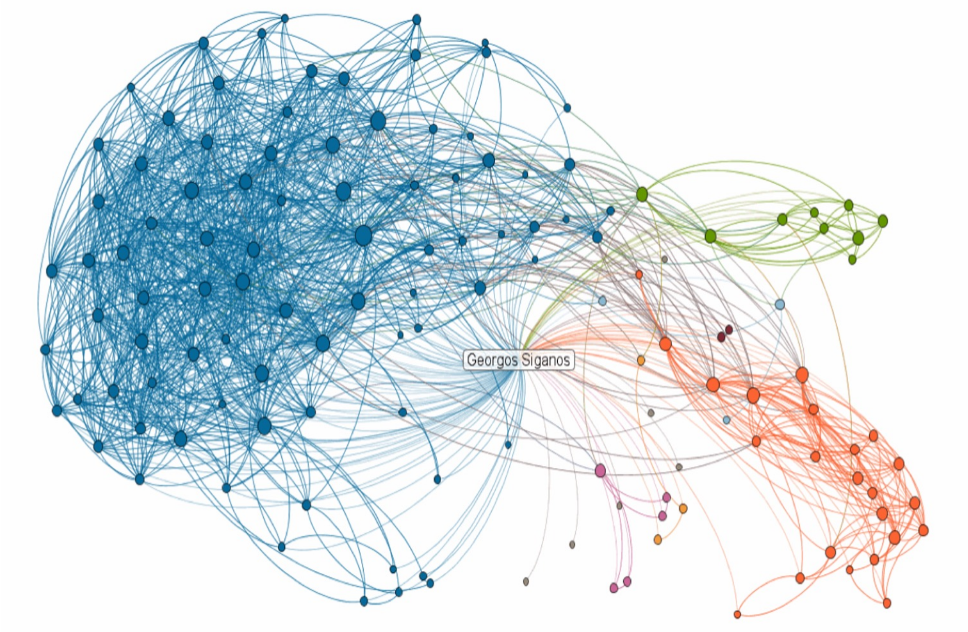
1. Why Graph Computing?
2. Pregel, GraphLab, PowerGraph
3. Ligra, GraphIt
4. Summary

Analyzing Big Graphs

- Many modern applications:
  - web search results
  - recommender systems
  - influence determination
  - advertising
  - anomaly detection

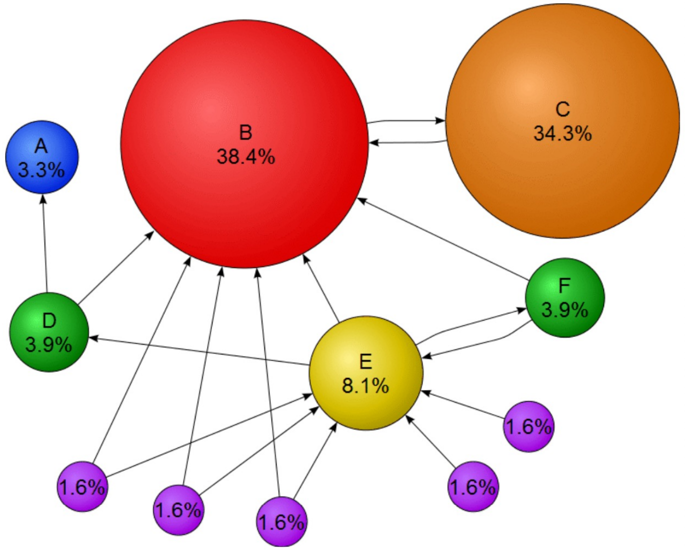
- Public dataset examples:
  - Twitter social graph
  - Wikipedia term occurrences
  - IMDB actors, Netflix
  - Amazon communities

Example graph computation: Page Rank

- Page Rank: iterative graph algorithm
- Graph nodes = web pages
- Graph edges = links between pages



$$\mathrm{PR}[v] = \beta + \alpha \sum_{u \in N^-(v)} \frac{\mathrm{PR}[u]}{deg^+(u)}$$

$$\alpha = 0.85, \ \beta = \frac{1-\alpha}{|V|}$$

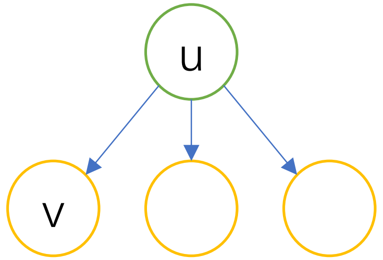
Rank of page v

Pages linked to page v

PageRank Example in C++ (Push mode)

$$\mathrm{PR}[v] = \beta + \alpha \sum_{u \in N^-(v)} \frac{\mathrm{PR}[u]}{deg^+(u)}$$

```
void pagerank(Graph &g, double * pr, double * new_pr, int max_iter) {
    for (int iter = 0; iter < max_iter; iter++) {
        for (vertex u : g.V()) {
            double temp = pr[u] / g.out_degree[u];
            for (vertex v : g.out_neighbors(u))
                new_pr[v] += temp;
        }
        for (vertex v : g.V()) {
            new_pr[v] = β + α * new_pr[v];
            pr[v] = new_pr[v]; new_pr[v] = 0;
        }
    }
}
```



Push mode

Hand-Optimized PageRank in C++

```
template<typename APPLY_FUNC>
 void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
     int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
     parallel_for(int n = 0; n < numVertices; n++) {
         for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
             local_new_rank[socketId][n] = new_rank[n]; } }
     int numPlaces = omp_get_num_places();
     int numSegments = g.getNumSegments("s1");
     int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
         #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
         int socketId = omp_get_place_num();
         for (int i = 0; i < segmentsPerSocket; i++) {
             int segmentId = socketId + i * numPlaces;
             if (segmentId >= numSegments) break;
             auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
             #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                 #pragma omp for schedule(dynamic, 1024)
                 for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                     NodeID d = sg->graphId[localId];
                     for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId + 1]; ngh++) {
                         NodeID s = sg->edgeArray[ngh];
                         local_new_rank[socketId][d] += contrib[s]; }}}}}
     parallel_for(int n = 0; n < numVertices; n++) {
         for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
             new_rank[n] += local_new_rank[socketId][n]; }}}
 struct updateVertex {
     void operator() (NodeID v) {
         double old_score = old_rank[v];
         new_rank[v] = (beta_score + (damp * new_rank[v]));
         error[v] = fabs((new_rank[v] - old_rank[v])) ;
         old_rank[v] = new_rank[v];
         new_rank[v] = ((float) 0) ; }; };
 void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
     for (int i = (0); i < (max_iter); i++) {
         parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
              contrib[v] = (old_rank[v] / out_degree[v]);};
         edgeset_apply_pull_parallel(edges, updateEdge());
         parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
              updateVertex()(v_iter); }; }
```

More than 23x faster

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores

Multi-Threaded
Load Balanced
NUMA Optimized
Cache Optimized

(1) Hard to write correctly

(2) Extremely difficult to experiment with different combinations of optimizations

Graph Processing Challenges

- Sparsity $\longrightarrow$ poor locality
- High communication-to-computation ratio
- Varying parallelism, race conditions, load imbalance

Can we build a Graph Processing System to handle these challenges?

Running time efficiency
Space efficiency
Programming efficiency

Interface between System and Programmer

- What tasks does the **system** take off the hands of the programmer?
  - tasks challenging or tedious enough?

- What tasks does the system leave to the **programmer**?
  - likely because the programmer is better at these tasks

System Tradeoff for *High Performance* & *High Productivity*

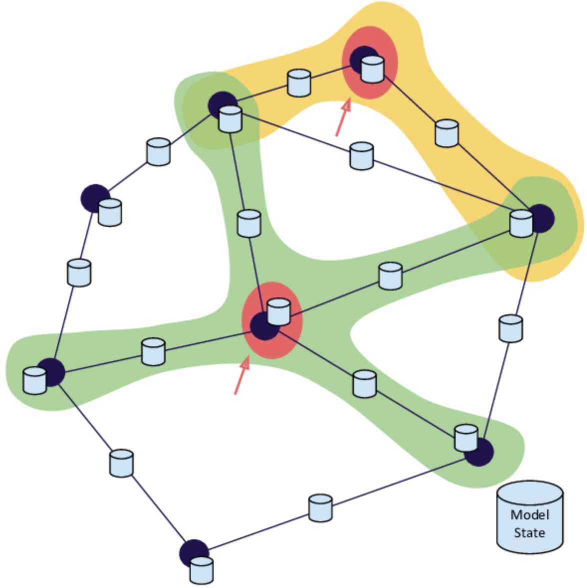
- What are the **fundamental operations** (i.e., **primitives**) ?
  - easy to express and efficient to execute

- What are the **key optimizations** in best implementations?
  - high-level abstractions should not prevent optimizations
  - Ideally even done by system automatically

Pregel

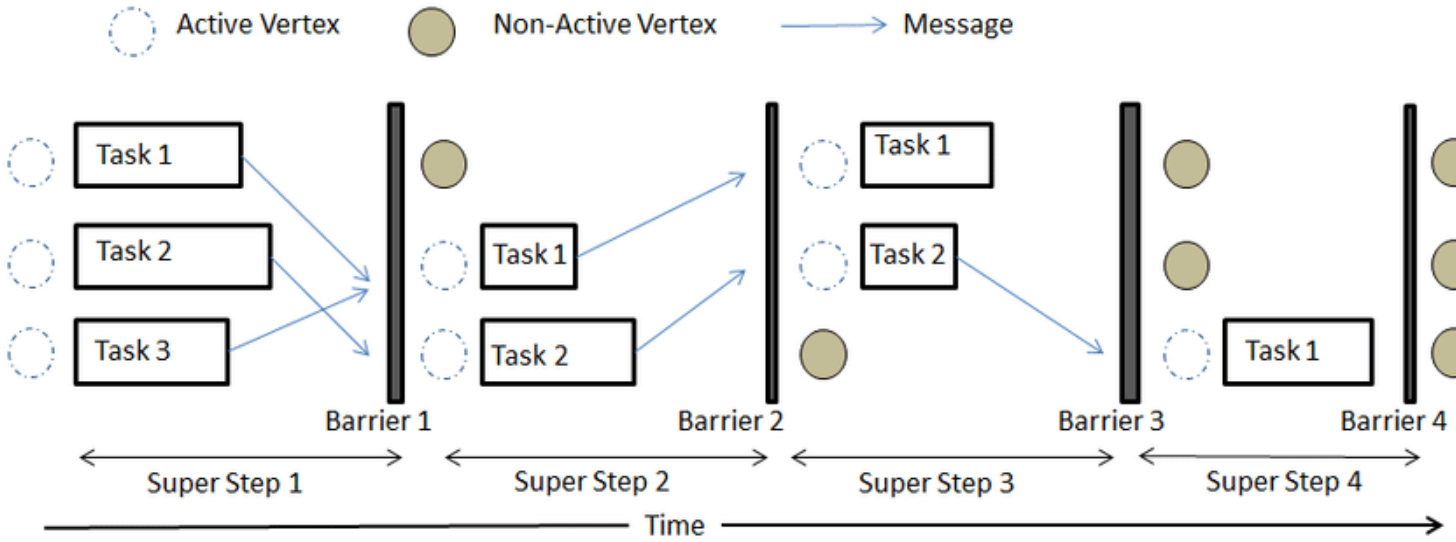
A System for Large-Scale Graph Processing

Pregel "Think like a vertex"



- **Vertex Program**: defines **update** on each **active** vertex
- **Bulk synchronous** model
- Distributed-memory, uses message passing

Tasks vary in size
→ Load imbalance

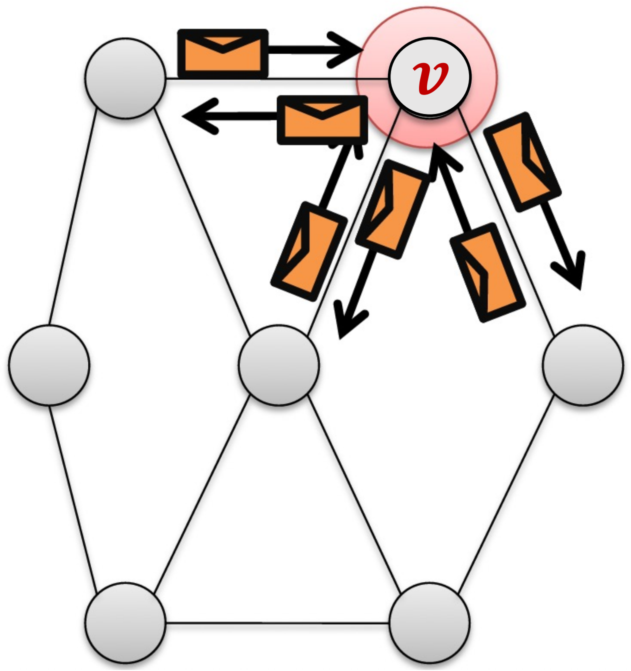


PageRank in Pregel

$$\mathrm{PR}[v] = \beta + \alpha \sum_{u \in N^{-}(v)} \frac{\mathrm{PR}[u]}{deg^{+}(u)}$$

Programmer's responsibility

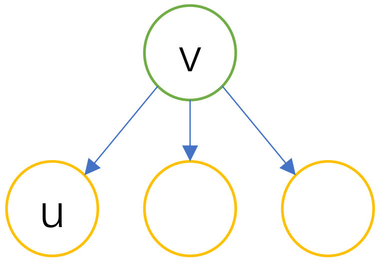
- **Vertex-Programs** interact by sending **messages**




```
Pregel_PageRank (vertex v, Message* messages) :
    // Receive all messages from the previous step
    double sum = 0;
    foreach (mesg in messages) :
        sum += mesg;
    // Update the rank of this vertex
    PR[v] = beta + alpha * sum;
    // Send messages to outgoing neighbors
    foreach (u in out_neighbors[v]) :
        Send  mesg(PR[v] / out_degree[v]) to vertex u
```

Sequential

do this for the next superstep



Push mode

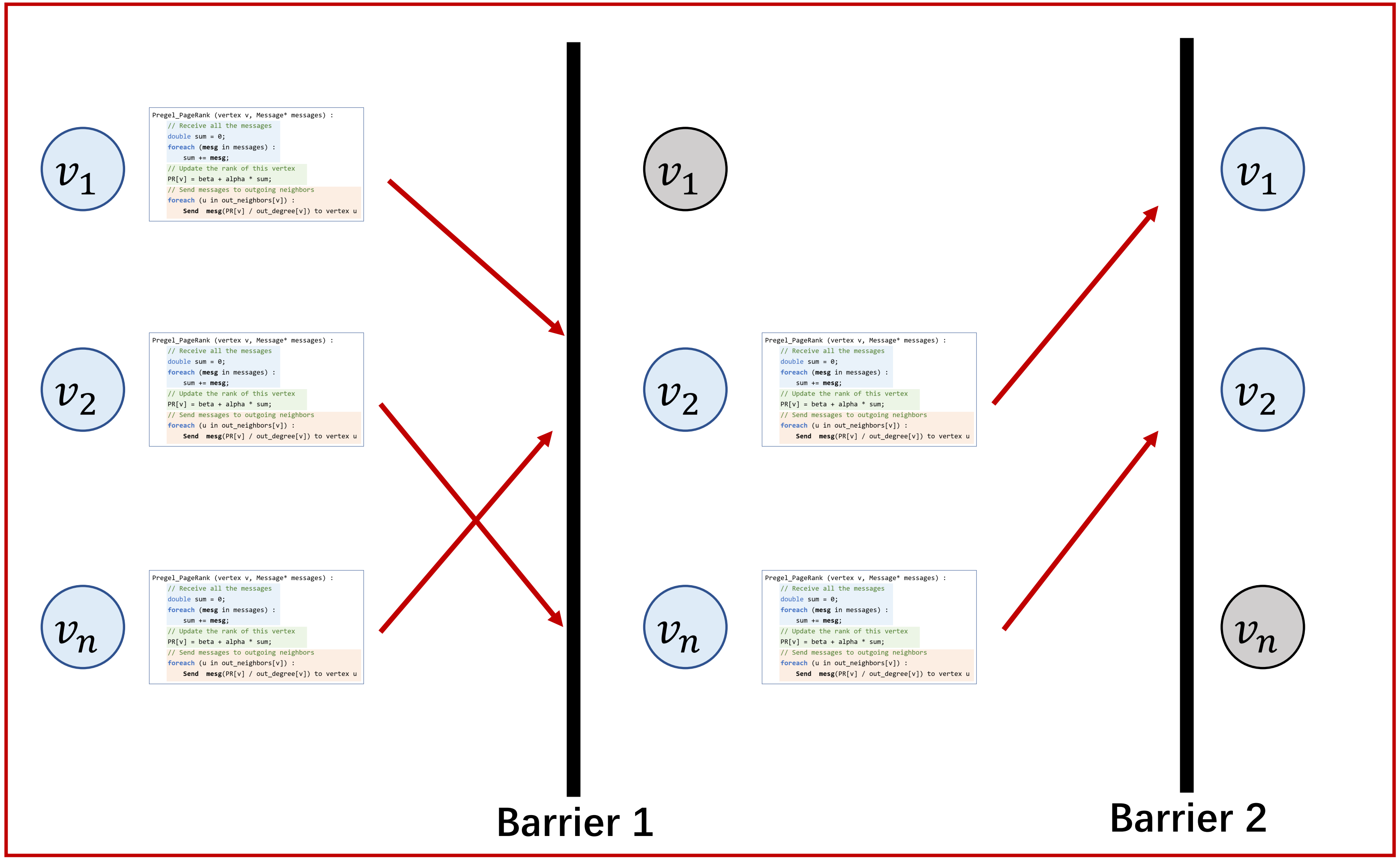
System's Responsibility

Pregel System's responsibility:

- Call the vertex-program on each active vertex;
- Implements communication among machines;
- Synchronous execution step-by-step

Programmer's responsibility: define the vertex-program

```
Pregel_PageRank (vertex v, Message* messages) :
    // Receive all the messages
    double sum = 0;
    foreach (mesg in messages) :
        sum += mesg;
    // Update the rank of this vertex
    PR[v] = beta + alpha * sum;
    // Send messages to outgoing neighbors
    foreach (u in out_neighbors[v]) :
        Send  mesg(PR[v] / out_degree[v]) to vertex u
```



Tradeoff

Primitive: V-program + msg. passing

Fixed exec. model → simple

No flexibility → lower performance

Pregel: Summary

Think like a vertex

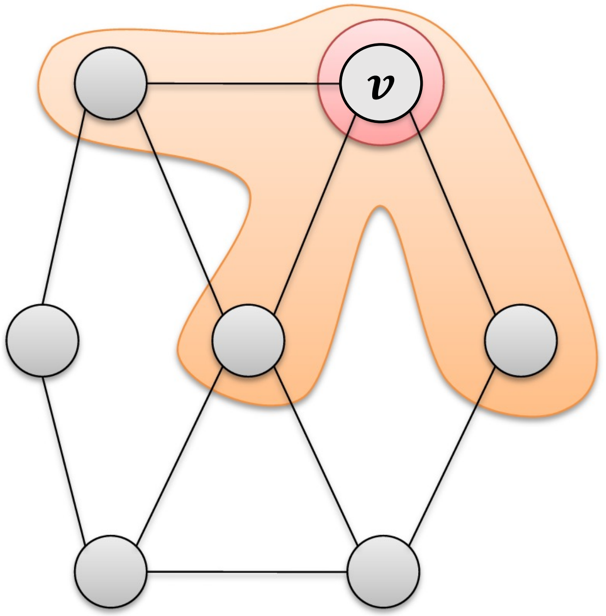
- Programmer defines a **vertex-program** that specifies
  - How to **update data** at each vertex
  - How to **communicate** (send/receive messages) with **neighbors**
- System is responsible for
  - Call the vertex-program (run it on distributed machines)
  - **Synchronously** execution $\longrightarrow$ simple $\longrightarrow$ **load imbalance**
  - Implement **communication** intra- or inter **machines** in a cluster
- Tradeoff: **simplicity** (productivity) vs. **flexibility** (performance)

GraphLab

A system for **asynchronous** graph computations

GraphLab[1] : **Asynchronous** graph computations

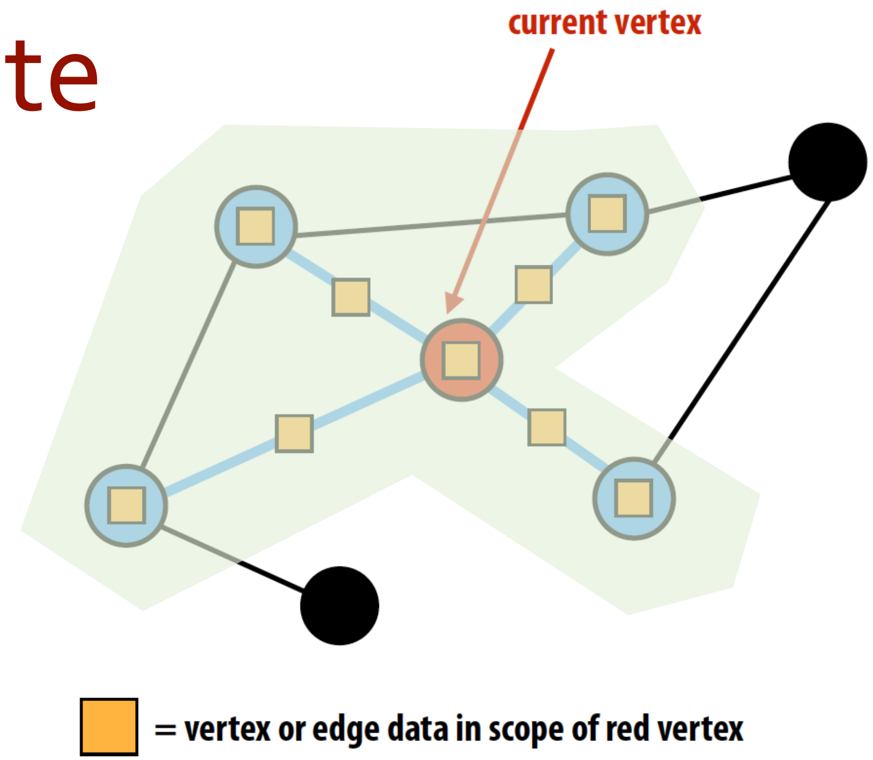
- **Think like a vertex**
- Vertex programs directly access neighbors' state
  - **Vertex-centric:** per-vertex update on the vertex's local neighborhood
  - **Shared-memory**: no message passing abstraction
  - **Asynchronous**: No barrier synchronization



[1] GraphLab: A New Framework for Parallel Machine Learning, Low et al. UAI 2010

The **vertex program**: local update

- Neighborhood (aka "**scope**") of vertex:
  - The current vertex
  - Adjacent edges
  - Adjacent vertices

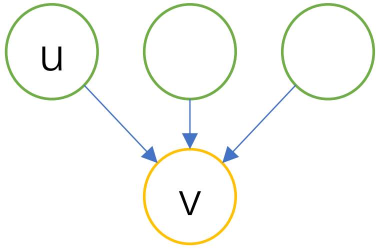


- Local Update function:
  - Defines *per-vertex operations* on a *scope* of a vertex: intuitive
  - No message passing abstraction
  - Uses **signaling** to create new tasks dynamically

PageRank in GraphLab

$$PR[v] = \beta + \alpha \sum_{u \in N^{-}(v)} \frac{PR[u]}{deg^{+}(u)}$$

- Vertex-Programs directly read the neighbors' state



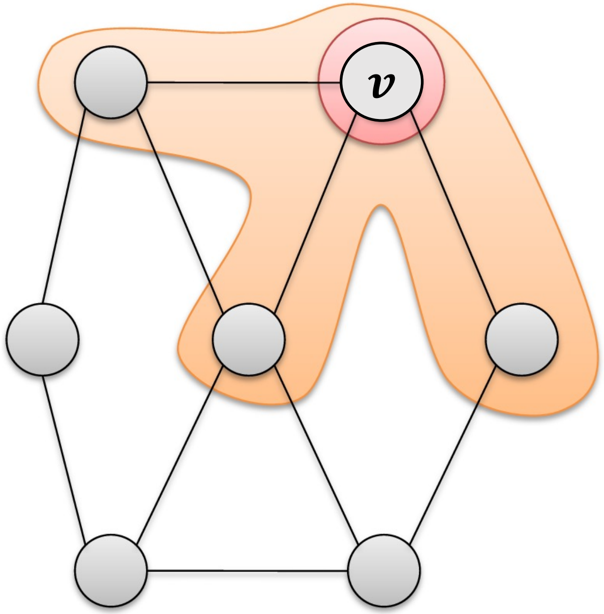
Pull mode

```
GraphLab_PageRank(vertex v) :
    // Compute the sum over neighbors
    sum = 0;
    foreach (vertex u in in_neighbors(v)) :
        sum += PR[u] / out_degree(u)

    // Update my rank (v)
    PR[v] = beta + alpha * sum;

    // Trigger neighbors to run again
    if PR[v] not converged then
      foreach(vertex u in out_neighbors(v)):
        signal vertex-program on u
```

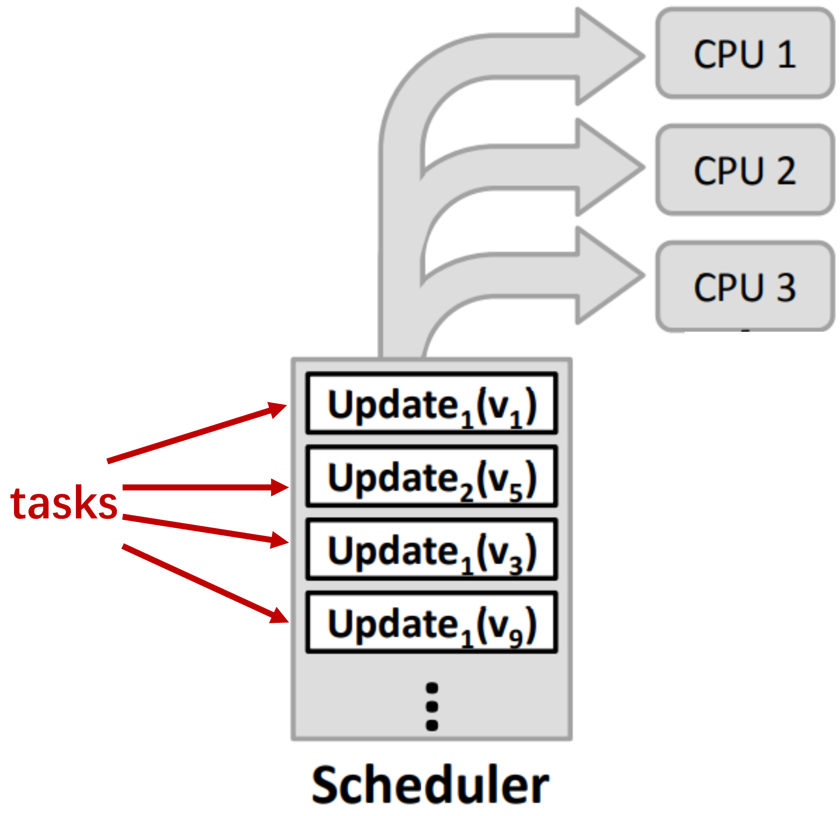
Sequential



Task Scheduling

- Each vertex-program on a vertex is a **task**
- GraphLab runtime is a task queue **scheduler**
- A *task scheduling policy* defines in which **order** that tasks are executed
  - scheduling **order** can be critical for performance or correctness/quality

```
GraphLab_runtime () :
    foreach (vertex v in task_queue) :
        call vertex_program(v)
```

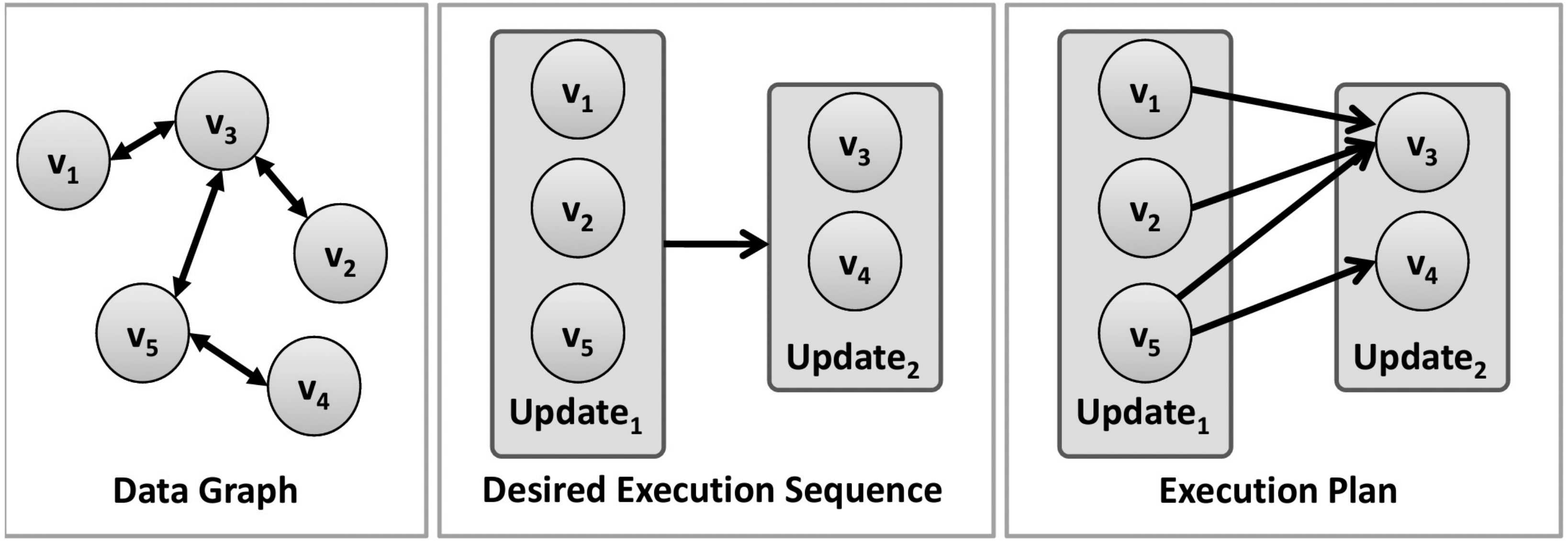


Task Scheduling

Tradeoff

Primitive: V-program + Scope + Signal
Opt: expose scheduler to programmer
better flexibility → **higher performance**
more complexity → **lower productivity**

- GraphLab provides a collection of basic schedulers
  - Synchronous: all in parallel
  - Round robin: all sequential
- Allows users to create their own scheduler
  - Asynchronous: user provides a data dependency graph → parallelize if no dependency



Summary: GraphLab

- The programmer defines local update at each vertex
  - directly access neighbors' data → more intuitive (no message passing)
  - can create work dynamically by **signal** → more efficient
- The system takes responsibility for scheduling and parallelization
  - support **asynchronous** execution model → no barrier synchronization
  - programmable scheduler → could be messy (blurs user/system interface)
- Tradeoff: **flexibility** (performance) vs. **complexity** (productivity)

PowerGraph

Distributed Graph Computation on Power-law Graphs

PowerGraph [1]: Optimizing for power-law graphs

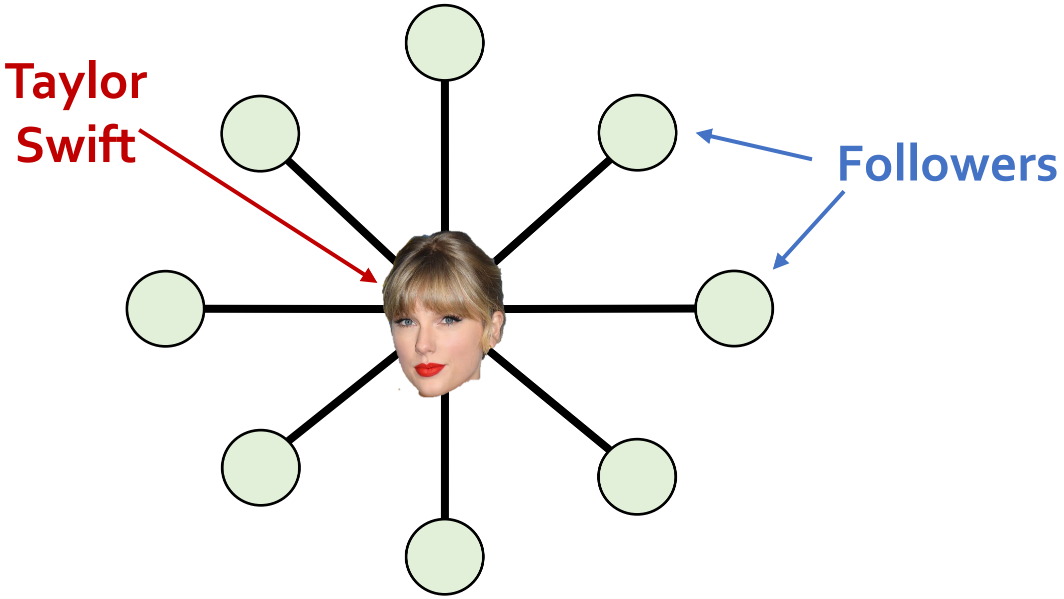
- High-degree vertices are problematic

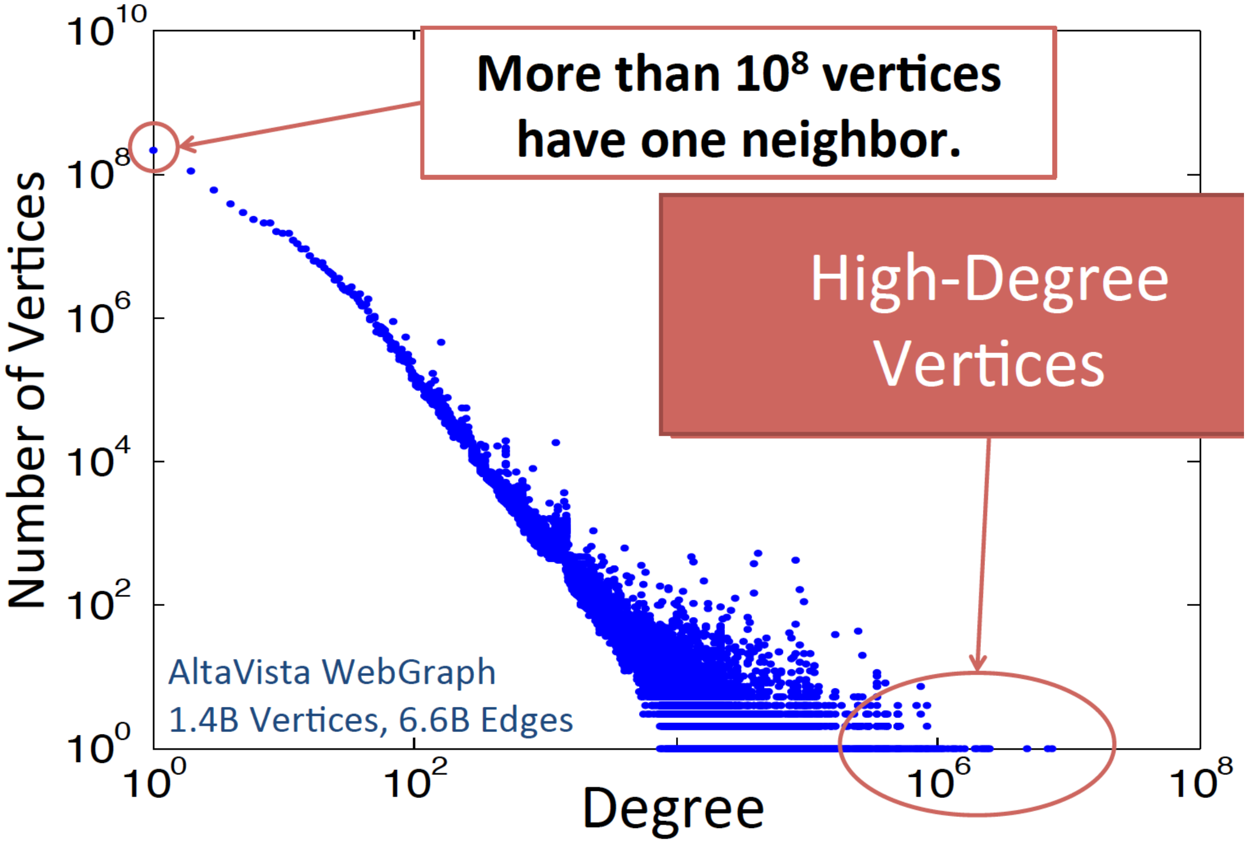
- Vertex program with GAS model
  - User defines **separated** Gather, Apply, and Scatter (GAS) functions

- GAS Decomposition enables optimizations
  - Split a single vertex-program over multiple machines
  - Parallelize high-degree vertices

[1] PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, Gonzalez et al. OSDI 2012

Real-world graphs: Power-Law Degree Distribution



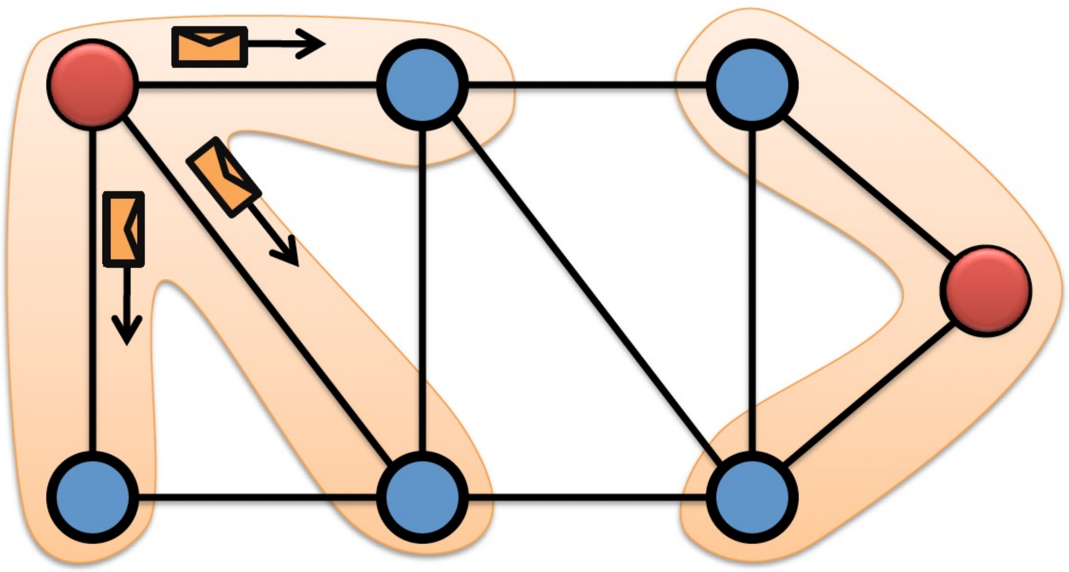


A small number of **high-degree vertices**
A large number of **low-degree vertices**

Challenges of High-Degree Vertices

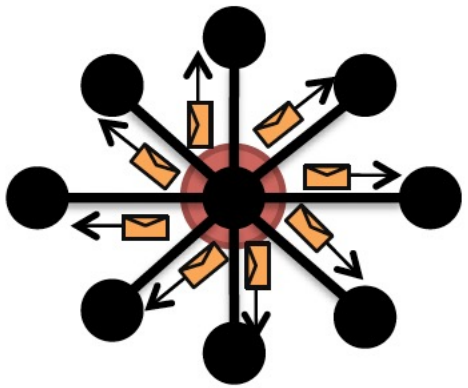
- A user-defined Vertex-Program runs on each vertex
  - Using messages, e.g., Pregel
  - Through shared state, e.g.,GraphLab

- **Vertex Parallelism**: run multiple vertex programs simultaneously

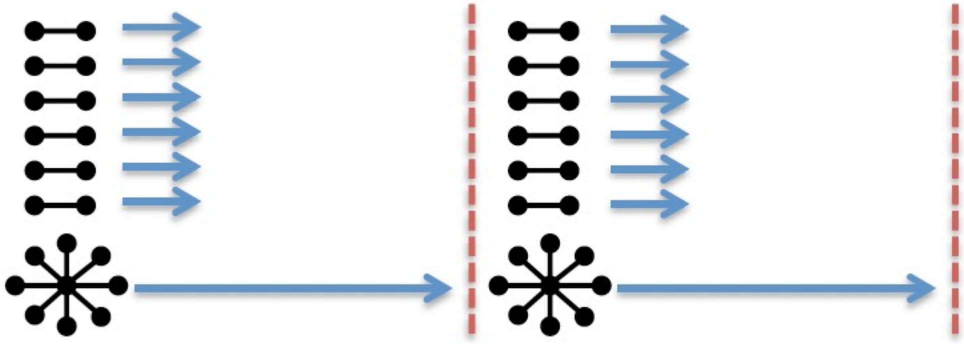


Challenges of High-Degree Vertices

Pregel

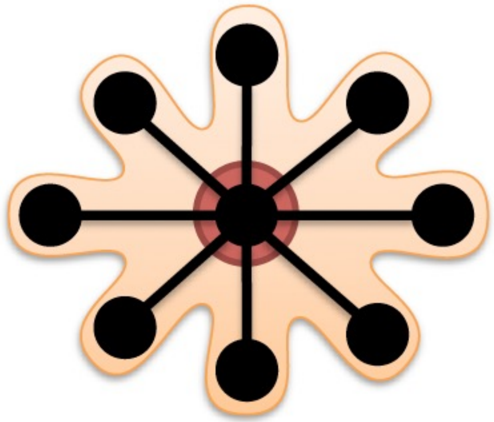


Sends many messages



Synchronous Execution prone to stragglers

GraphLab

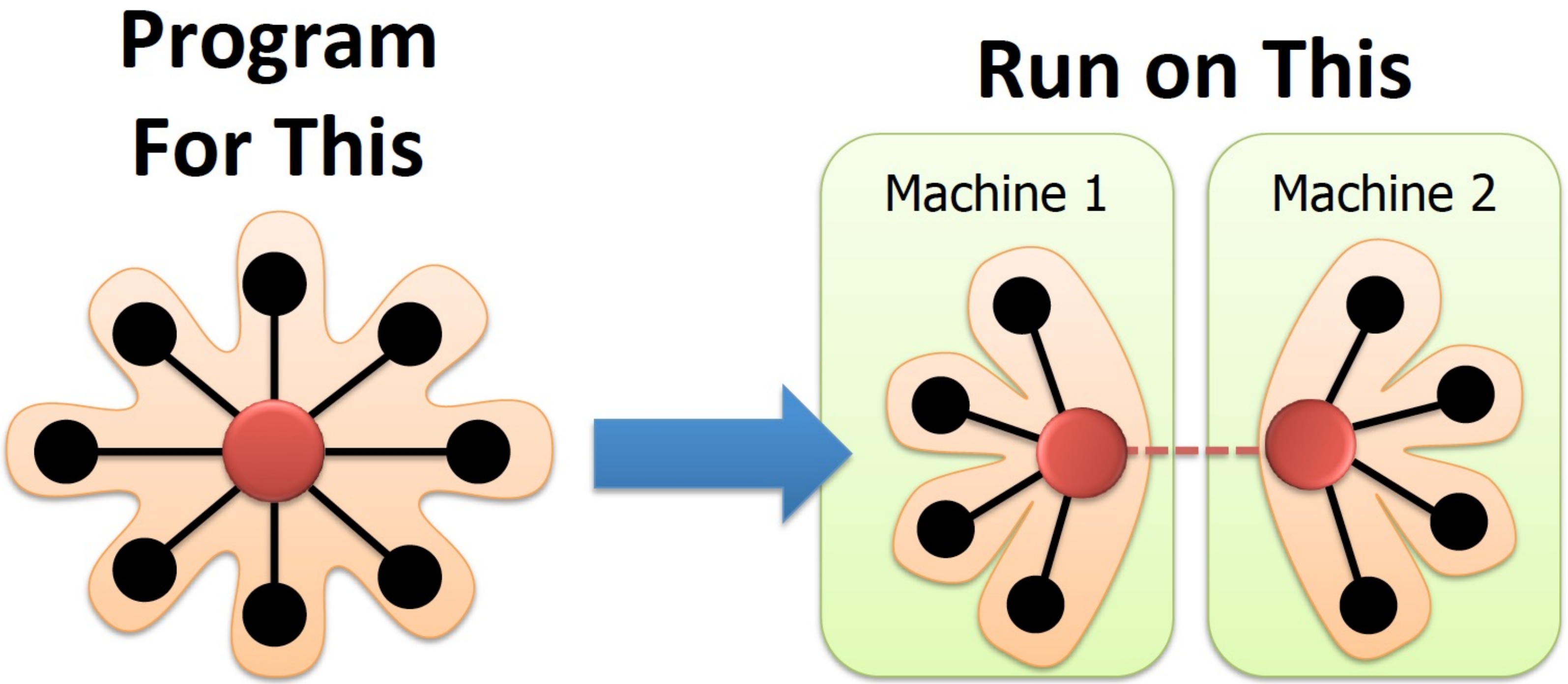


Touches a large fraction of graph

Asynchronous Execution requires heavy locking

A Solution: Split High-Degree vertices

- Split the task (edges) of a high-degree vertex across multiple machines



Edge Parallelism?

Can we do Split in Pregel or GraphLab?

```
GraphLab_PageRank(vertex v) :
    // Compute the sum over neighbors
    sum = 0;
    foreach (vertex u in in_neighbors(v)) :
        sum += u.rank / out_degree(u)

    // Update my rank (v)
    v.rank = beta + alpha * sum;

    // Trigger neighbors to run again
    if R[v] not converged then
      foreach(vertex u in out_neighbors(v)):
        signal vertex-program on u
```

Sequential

A Common Pattern for Vertex-Programs

Gather Information about Neighborhood

Update Vertex

Signal Neighbors & Modify Edge Data

PowerGraph: GAS Decomposition

Key idea: **Decompose** the vertex-program into three phases

```
GraphLab_PageRank(vertex v) :
   // Compute the sum over neighbors
   sum = 0;
   foreach (vertex u in in_neighbors(v)) :
       sum += PR[u] / out_degree(u)

   // Update my rank (v)
   PR[v] = beta + alpha * sum;

   // Trigger neighbors to run again
   if PR[v] not converged then
     foreach(vertex u in out_neighbors(v)):
       signal vertex-program on u
```

vertex-program

Gather

Apply

Scatter

user-defined functions

PageRank in PowerGraph

$$PR[v] = \beta + \alpha \sum_{u \in N^{-}(v)} \frac{PR[u]}{deg^{+}(u)}$$

```
PowerGraph_PageRank(v)

Gather(u → v) : return PR[u]/out_degree[u]

sum(a, b) :  return a + b;

Apply(v, Σ) : PR[v] = beta + alpha * Σ

Scatter(v → u) :
  if PR[v] changed then trigger u to be recomputed
```

No for loops!

PowerGraph System Runtime

fine-grained

```
PowerGraph_runtime* () :
    foreach (vertex v in task_queue) :
        // Compute the sum over neighbors
        Σ = 0;
        foreach (vertex u in in_neighbors(v)) :
            Σ = sum(Σ, call gather(u, v))

        // Update my rank (v)
        call apply(v, Σ)

        // Trigger neighbors to run again
        foreach (vertex u in out_neighbors(v)):
            call scatter(u, v)
```

for loops in system runtime

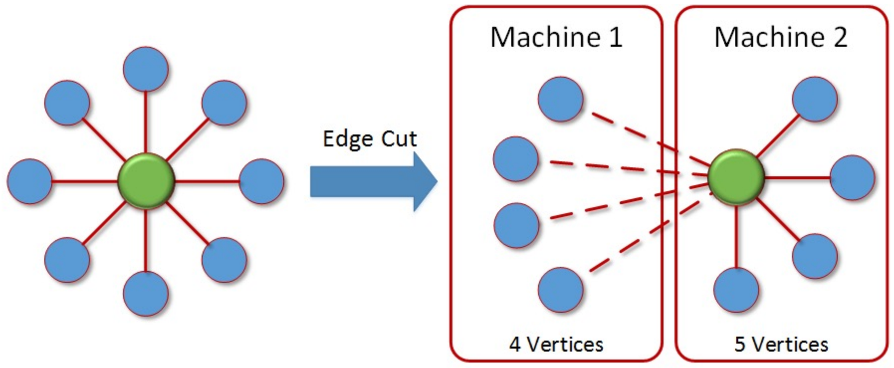
coarse-grained

```
GraphLab_runtime () :
    foreach (vertex v in task_queue) :
        call vertex_program(v)
```

* For simplicity, the pseudocode is only for a single iteration

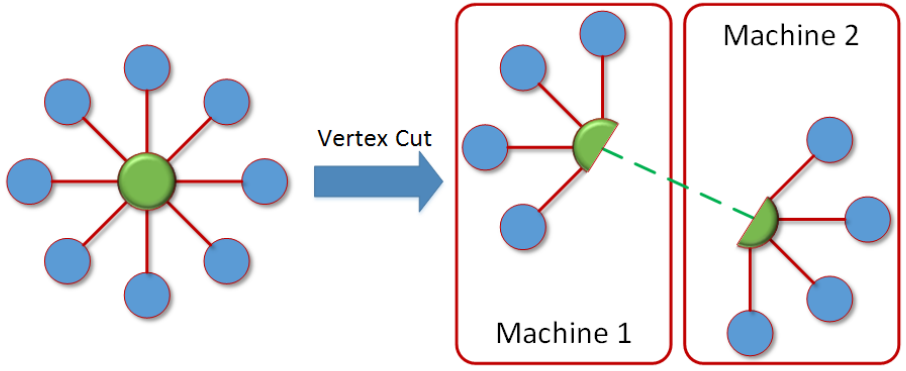
Graph Partitioning for Parallel Processing

Edge Cut



- Evenly assign **vertices** to machines
- Used by Pregel and GraphLab abstractions

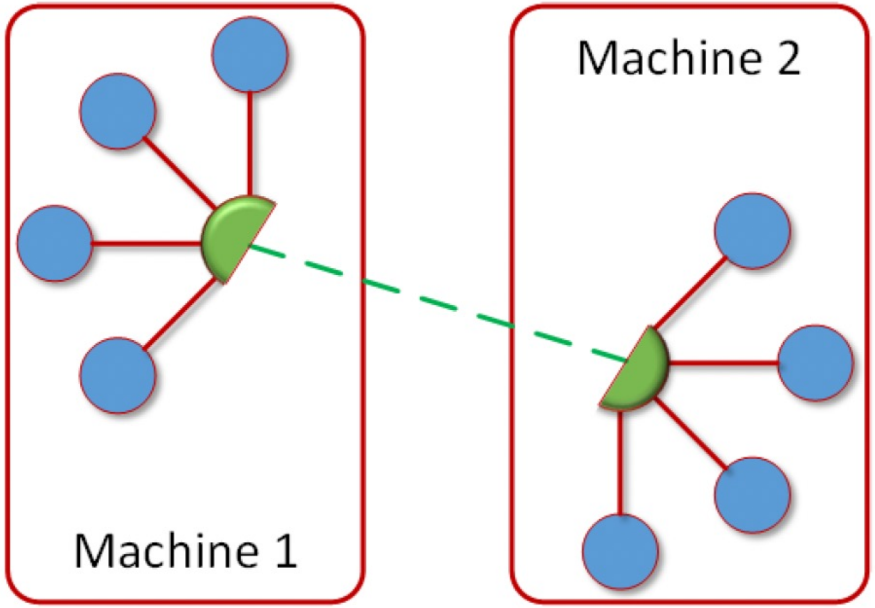
Vertex Cut



- Evenly assign **edges** to machines
- Used by PowerGraph abstraction

GAS Decomposition enables Vertex-Cut

- Vertex cut distributes a single vertex-program across multiple machines
- Allows to parallelize high-degree vertices



Tradeoff

Primitive: GAS Decomposition

Optimization: vertex-cut

improve parallelism → higher Performance

less flexibility?

Summary: PowerGraph

- Prior systems perform poorly on power-law graphs
  - High-degree vertices
  - Low-quality edge-cuts

- **Solution**: PowerGraph System Abstraction
  - **GAS Decomposition**: split vertex programs → enables **vertex-cut**
  - **Vertex-cut partitioning**: distribute natural graphs

- **Tradeoff**: GAS is a fine-grained model
  - Enables Split-vertex → more parallelism, better load balance
  - Not intuitive edge parallelism, Hard to enable some optimizations
  - We will see how this is solved in Ligra

Pregel

- Think like a vertex
- Vertex programs interact by sending **messages**
- **Synchronous** execution

GraphLab

- Vertex programs directly read neighbors' state
- **Asynchronous** execution
- Programmable task scheduler

PowerGraph

- **GAS Decomposition** (power-law)
- Vertex-cut partitioning
- Parallelize high-degree vertices

Any limitations of the "Think like a vertex" model?