Krill: A Compiler and Runtime System for Concurrent Graph Processing

Hongzheng Chen, Minghua Shen, Nong Xiao, Yutong Lu

Sun Yat-sen University

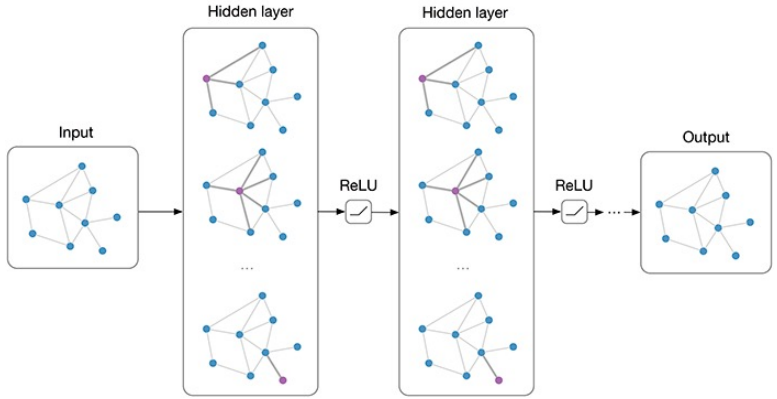
SC'21
Nov 17, 2021, St. Louis, MO

Graphs are everywhere …

Internet

Social Media

Sci. Comp.



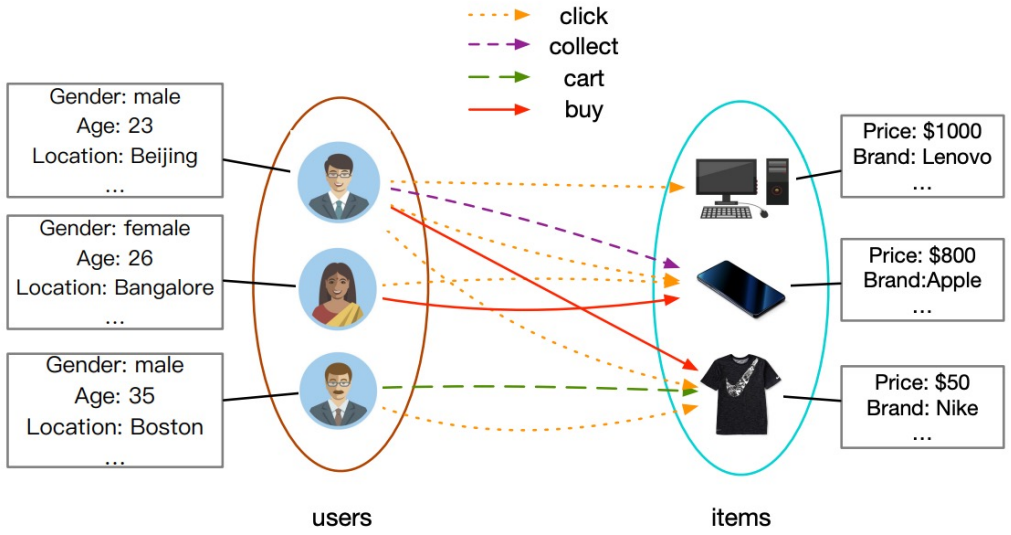
ML

Increasing Demand of Concurrent Graph Processing

Scenario 1: Mountains of concurrent graph query requests on the cloud

- Multiple queries are issued by different users simultaneously on the **same** graph

LinkedIn social network
Query: Degree of connection



Alibaba user-product graph
Query: Customized recommendation

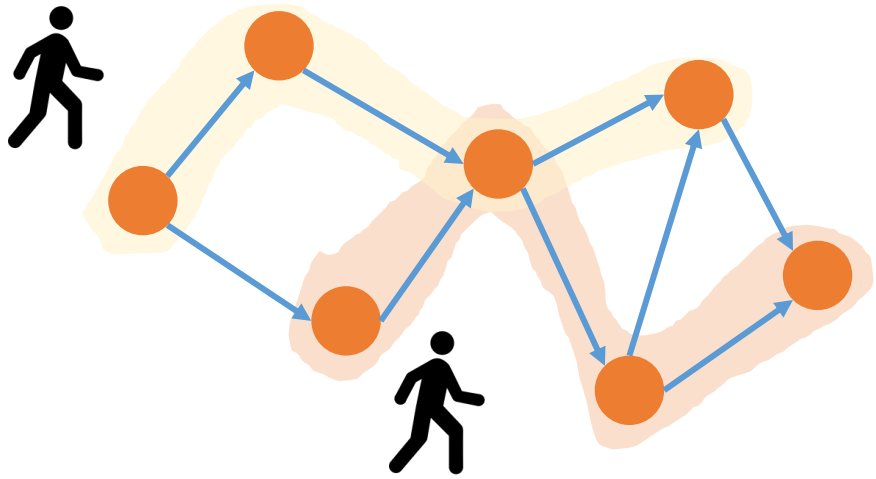
Require low response latency!

Fig source: 1. Microsoft, https://appsource.microsoft.com/es-es/product/power-bi-visuals/WA104381236?tab=Overview
2. Rong Zhu, Kun Zhao, Hongxia Yang et al. AliGraph: a comprehensive graph neural network platform. VLDB. 2019.

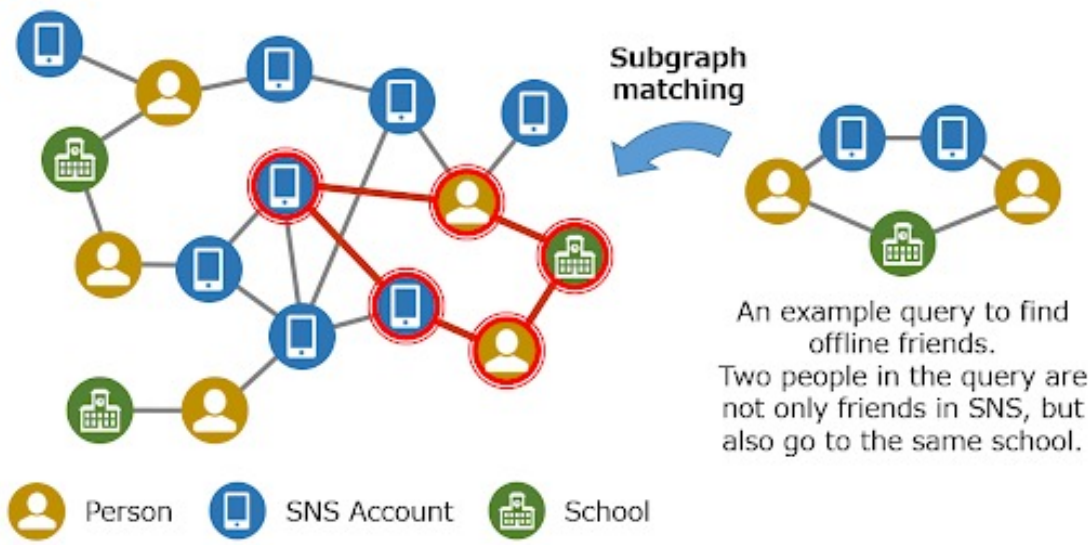
Increasing Demand of Concurrent Graph Processing

Scenario 2: Complex graph analytic applications

- A wide range of graph applications are composed of several basic graph algorithms



Random Walk (w/ multiple walkers)



Subgraph Matching (different patterns)

Require high job parallelism!

* Fig source: Onizuka Lab. http://www-bigdata.ist.osaka-u.ac.jp/en/research/

Challenges of Concurrent Graph Processing

Most of the existing graph systems are designed for processing a single job

```
struct BFS_F {
  BFS_F(uintE* _Parents) : Parents(_Parents) {}
  inline bool update (uintE s, uintE d) {
    if(Parents[d] == UINT_E_MAX) { Parents[d] = s; return 1; }
    else return 0;}
  inline bool updateAtomic (uintE s, uintE d){
    return (CAS(&Parents[d],UINT_E_MAX,s));
  }
  inline bool cond (uintE d) { return (Parents[d] == UINT_E_MAX); }
  uintE* Parents;
};

void Compute(graph& G) {
  long n = G.n;
  uintE* Parents = newA(uintE,n);
  parallel_for(long i=0; i<n; i++) Parents[i] = UINT_E_MAX;
  Parents[start] = start;
  vertexSubset Frontier(n,start);
  while(!Frontier.isEmpty()){
    vertexSubset output = edgeMap(G, Frontier, BFS_F(Parents));
    Frontier.del();
    Frontier = output;}
  Frontier.del();
  free(Parents);
}
```

Example: BFS written in Ligra

Key Problem: Tightly coupled graph model

- Algorithm：Specific compute patterns, e.g., BFS, SSSP
- Structure：Connectivity between vertices, usually stored in CSR format
- Property：Affiliated feature values e.g., Parents in BFS, PR value in PageRank

When processing concurrent graph jobs:

- Cannot share underlying structure across jobs
- Cannot make cross-job property optimization

↓

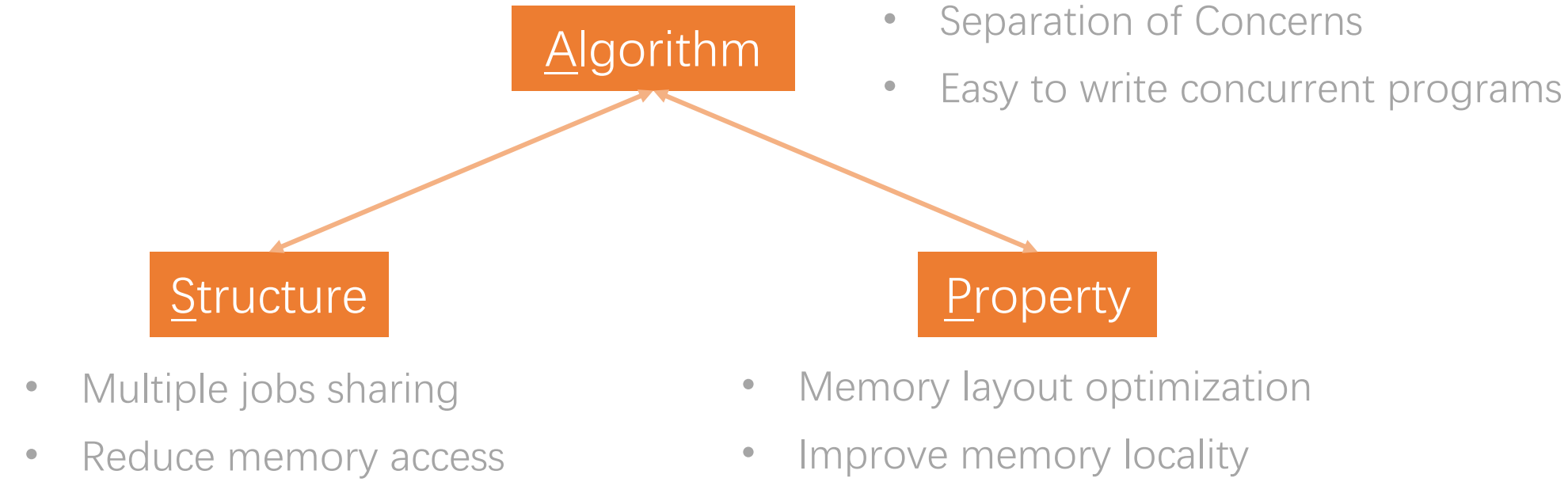
Large compute & memory access overhead

* SHUN J, BLELLOCH G E. Ligra: A Lightweight Graph Processing Framework for Shared Memory. PPoPP. 2013.

Decoupled SAP Graph Model

Decouple them!

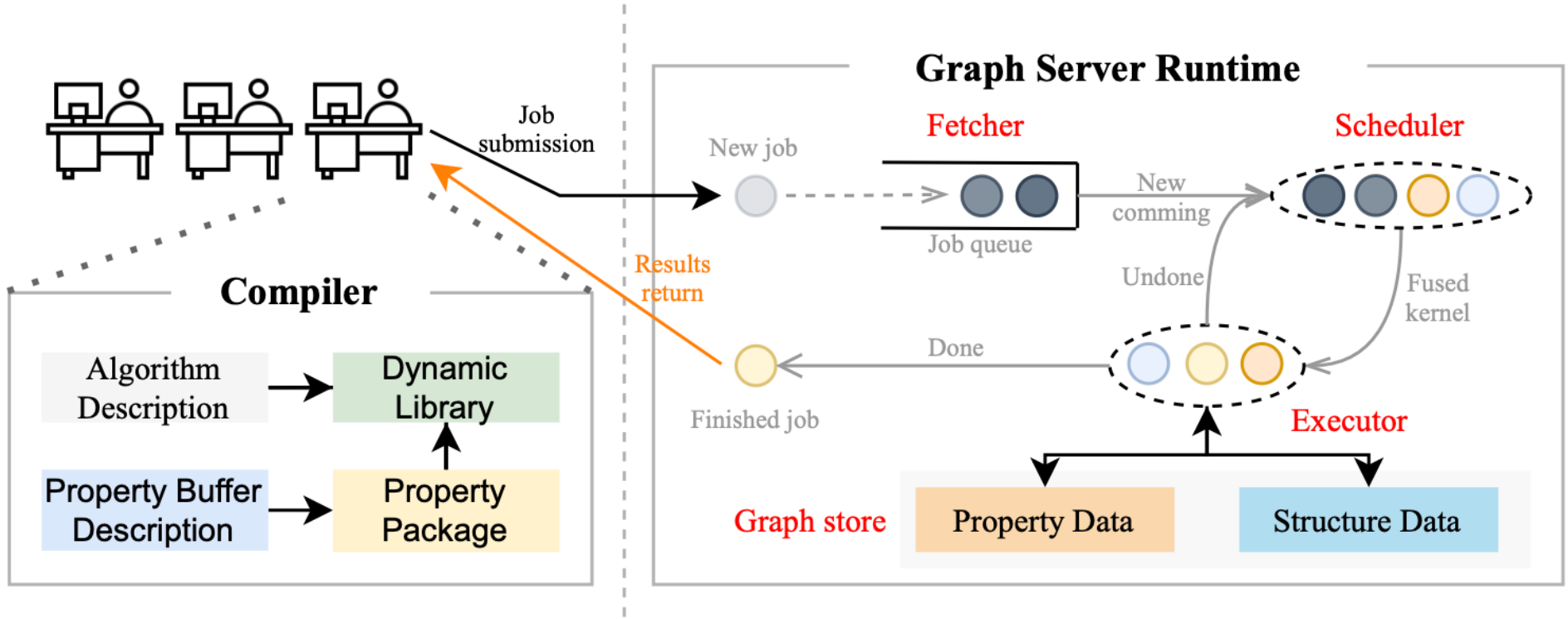
- **Graph algorithm**: The computation patterns
- **Graph structure**: Edge connectivity
- **Graph property**: Property data attached in each vertex



Krill: Concurrent Graph Processing System

Compiler + Runtime

- **Property buffers** for easy property data declaration & management
- **Graph kernel fusion** for efficient graph structure traversal & computation



Challenge 1: Missing Property Management

Manually create property data when writing the graph algorithm

- Different types of data are hard to write and organize
  - Mixing types in a class involves memory alignment
  - Cause extra memory usage (more serious when CG jobs are considered)

- The system has no control on these property data
  - Scattered around the memory -> Bad temporal memory locality
  - e.g. PageRank

$$pr_{next}[dst] \mathrel{+}= \frac{pr_{curr}[src]}{outneighbor(src)}$$

  - Prevent optimization across different jobs

Property Buffers

- Declare property data in a separated file (named .propbuf)
- Protocol-buffer-like grammar <type><name> = <init_val>;
- Better organize property data mixing different types
- Compile to a header file for further linkage

Demo.propbuf

```
property BFS {
   int Parents = -1;
}

property SSSP {
   int MinLen = MAX_INT;
}
```

Source-to-source Compilation

Demo.pb.h

```
class Parents { // BFS_Prop
   int get (int i) const;
   void set (int i, int val);
   // more accessors
};
class MinLen { // SSSP_Prop
   int get (int i) const;
   void set (int i, int val);
   // …
};
class PropertyManager {
   PropertyManager (size_t n);
   BFS_Prop::Parents* add_Parents();
   SSSP_Prop::MinLen* add_MinLen();
   void initialize();
};
```

* Google Protocol Buffer, https://developers.google.com/protocol-buffers

Property Buffers

- Easy property access using **.get() /.set()** methods
- Add compute kernels (jobs) by setKernels(G, K)

Property Declaration

Demo.propbuf

```
property BFS {
   int Parents = -1;
}
```

```
property SSSP {
   int MinLen = MAX_INT;
}
```

Algorithm Description

BFS.h (pseudocode)

```
BFS(size_t n, PropertyManager& prop, int start):
   parents = prop.add_Parents();
cond(dst): return (parents->get(dst) == -1);
update(src, dst):
   if (parents->get(dst) == -1)
      parents->set(dst, src);
```

SSSP.h (pseudocode)

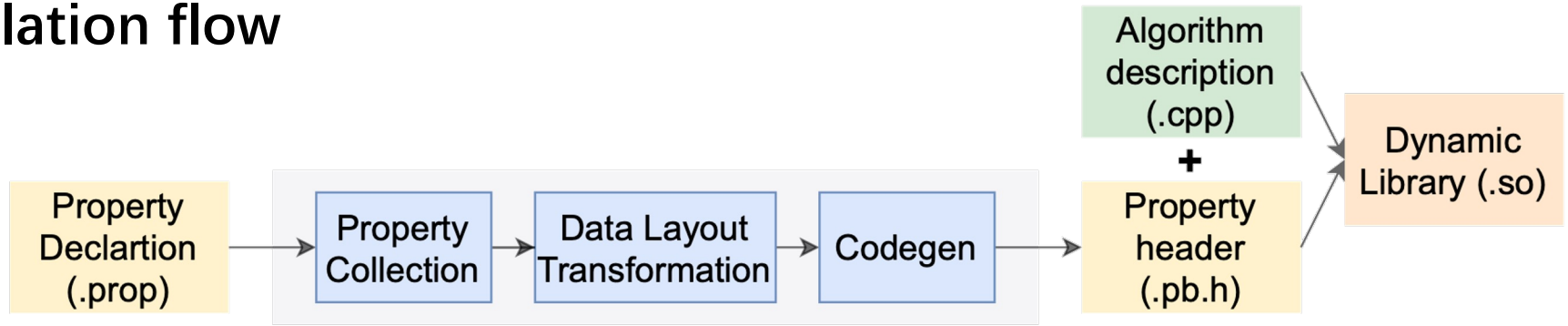
```
SSSP(size_t n, PropertyManager& prop, int start):
   minLen = prop.add_MinLen();
cond(dst): return true;
update(src, dst, edgeLen):
   newLen = minLen->get(src) + edgeLen;
   if (minLen->get(dst) > newLen)
      minLen->set(dst, newLen);
```

Demo.cpp

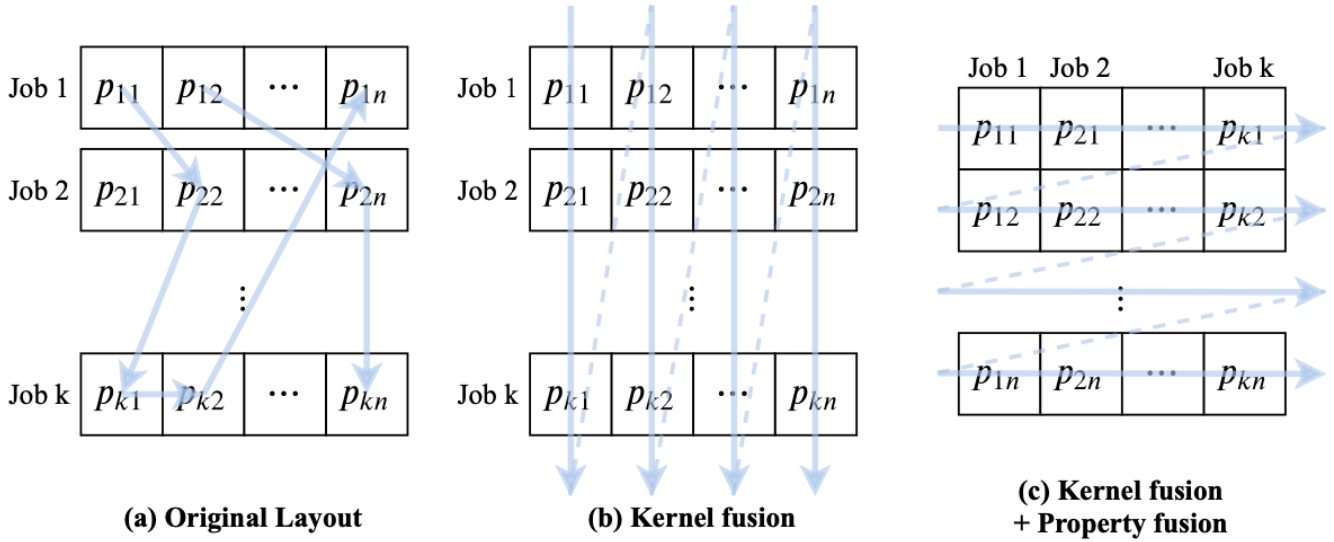
```
void setKernels(Graph& G, Kernels& K)
{
   Demo::PropertyManager prop(G.n);
   BFS* bfs = new BFS(G.n, prop, 0);
   SSSP* sssp = new SSSP(G.n, prop, 0);
   K.appendJob({bfs, sssp});
   prop.initialize();
}
```

Property Buffer Compiler

- **Compilation flow**

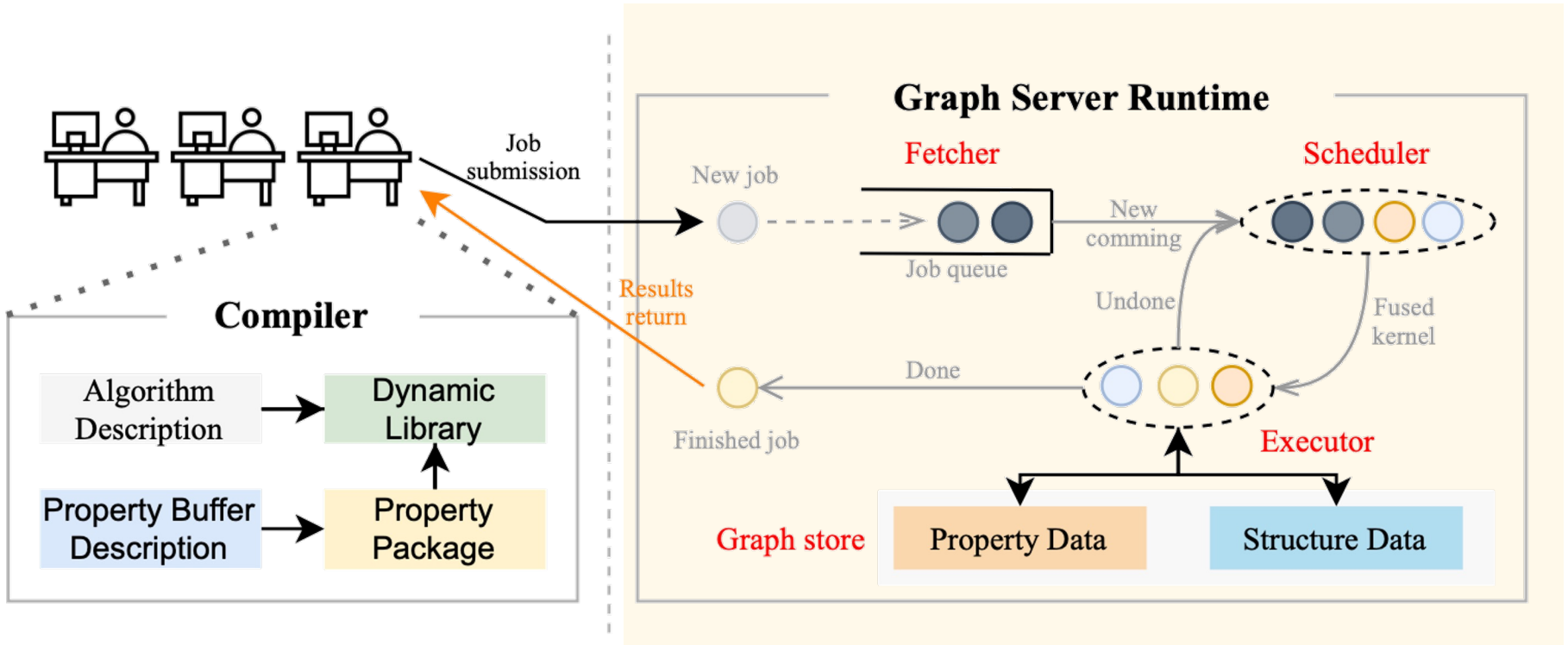


- Can make **data layout transformation** that organizes data of different jobs together to improve **spatial locality**



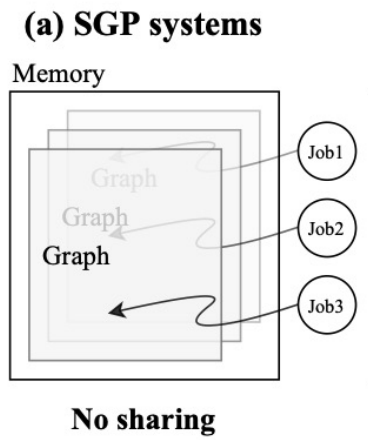
Runtime System

- **Graph store**: Long-running service storing structure & property data in memory
- **Fetcher**: Fetch the new-coming jobs and put into job queue
- **Scheduler**: Manipulate graph kernel fusion
- **Executor**: Execute specific graph algorithm



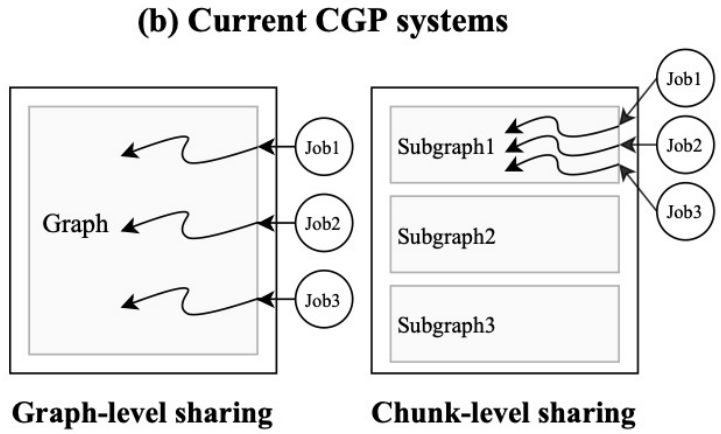
Challenge 2: Redundant Structure Accesses

- **Existing Single Graph Job Processing (SGP) systems**
  - **Ligra[PPoPP'13], Gemini[OSDI'16]**
  - Multiple copies of graph structure



- **Existing Concurrent Graph Jobs Processing (CGP) system**
  - **Seraph[HPDC'14], CGraph[ATC'18], GraphM[SC'19]**
  - Shared graph structure, but jobs still access the graph individually

- Can further reduce accesses by exploring
  - Similar graph traversal patterns
  - Shared active vertices



Graph Kernel Fusion

- **Existing CGP systems require $O(c|E|)$ times of structure access**

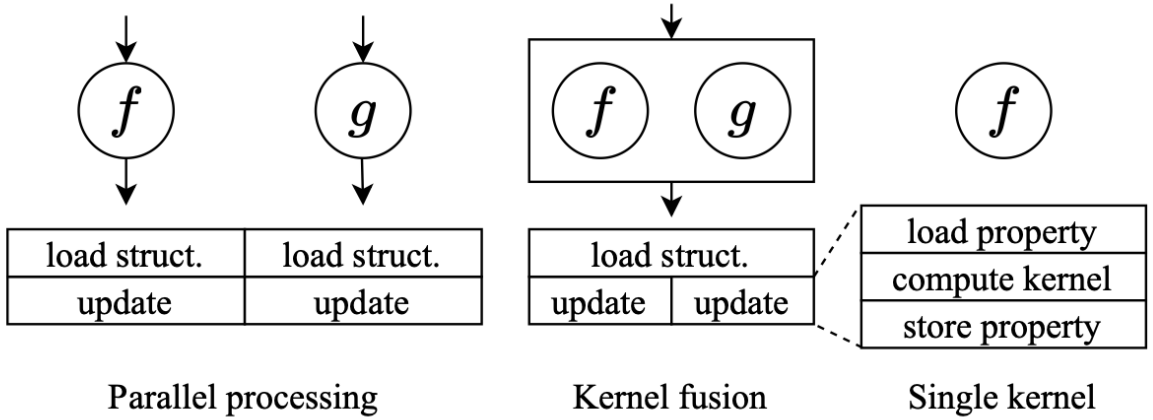
(a) Basic Push engine in CGP

```
1 for job in jobs do
2     for each vSrc in vertices do
3         if (!vSrc in frontier) continue
4         for vDst in vSrc.outngh do
5             if cond(vDst) then
6                 updateAtomic(vSrc, vDst)
```

(b) Basic Pull engine in CGP

```
1 for job in jobs do
2     for each vDst in vertices do
3         if (!cond(vDst)) continue
4         for vSrc in vDst.inngh do
5             if vSrc in frontier then
6                 update (vSrc, vDst)
```

- **Key idea: Force all the jobs to access the same edge, only $O(|E|)$ access!**



Graph Kernel Fusion

- **Moving in the job loop leads to $O(|E|)$ times of structure access**

(a) Modified Push engine in CGP

```
1 for each vSrc in vertices do
2    if (!vSrc in frontier) continue
3    for vDst in vSrc.outngh do
4        if cond(vDst) then
5            for job in jobs do
6                updateAtomic(vSrc, vDst)
```

(b) Modified Pull engine in CGP

```
1 for each vDst in vertices do
2    if (!cond(vDst)) continue
3    for vSrc in vDst.inngh do
4        if vSrc in frontier then
5            for job in jobs do
6                update (vSrc, vDst)
```

Graph Kernel Fusion

- **However, vertices cannot be skipped in the outer loop to ensure correctness**
- **Exactly $O(|E|)$ access in one iteration**

(a) Modified Push engine in CGP

```
1 for each vSrc in vertices do
2   if (!vSrc in frontier) continue
3   for vDst in vSrc.outngh do
4     if cond(vDst) then
5       for job in jobs do
6         if vSrc & vDst meet requirement
7           updateAtomic(vSrc, vDst)
```

(b) Modified Pull engine in CGP

```
1 for each vDst in vertices do
2   if (!cond(vDst)) continue
3   for vSrc in vDst.inngh do
4     if vSrc in frontier then
5       for job in jobs do
6         if vSrc & vDst meet requirement
7           update (vSrc, vDst)
```

Graph Kernel Fusion w/ Fast Filters

Can do better with two filters! ($\ll O(|E|)$ access)

- **Common Frontier Filer**: Keep the union of frontiers to filter out active vertices
- **Advanced Job Filter**: Examine if the dst vertex of the job is active

(a) Common Frontier Filter

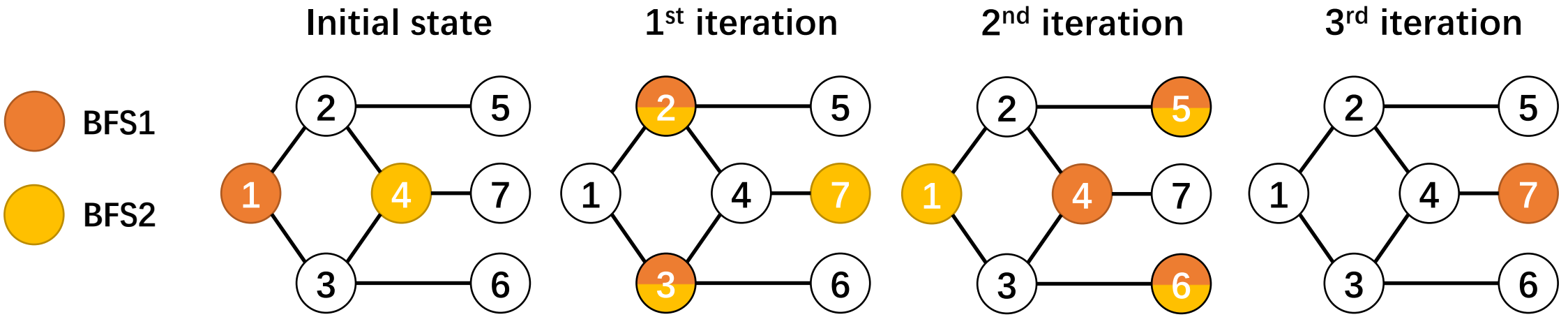
```
1 for each vSrc in comonFrontier do
2     for vDst in vSrc.outngh do
3         for job in jobs do
4             if vSrc & vDst meet requirement
5                 … // update job
```

(b) Advanced Job Filter

```
1 for each vDst in vertices do
2     … // calculate availJob
3     for vSrc in vDst.inngh do
4         for job in availJob do
5             if vSrc & vDst meet requirement
6                 … // update job
```

Graph Kernel Fusion w/ Fast Filters

- Traverse graph structure **once** but do multiple computation for each vertex
- Greatly reduce # of memory accesses
- Two BFS example:



GraphM [SC'19]	visited: {1}, {4}	visited: {2, 3}, {2, 3, 7}	visited: {1, 5, 6}, {4, 5, 6}	visited: {7}, {}
Krill	visited: {1, 4}	visited: {2, 3, 7}	visited: {1, 4, 5, 6}	visited: {7}

Common Frontier Filter

Advanced Job Filter

Experimental Setup

- Build Krill on top of Ligra (shared-mem) & Gemini (dist)
- Baseline systems
  - Ligra-S: Sequentially execute jobs, still parallel in one job
  - Ligra-P: Execute jobs in parallel
  - GraphM: The state-of-the-art shared-memory CGP system
  - Seraph: The state-of-the-art distributed CGP system
- Datasets

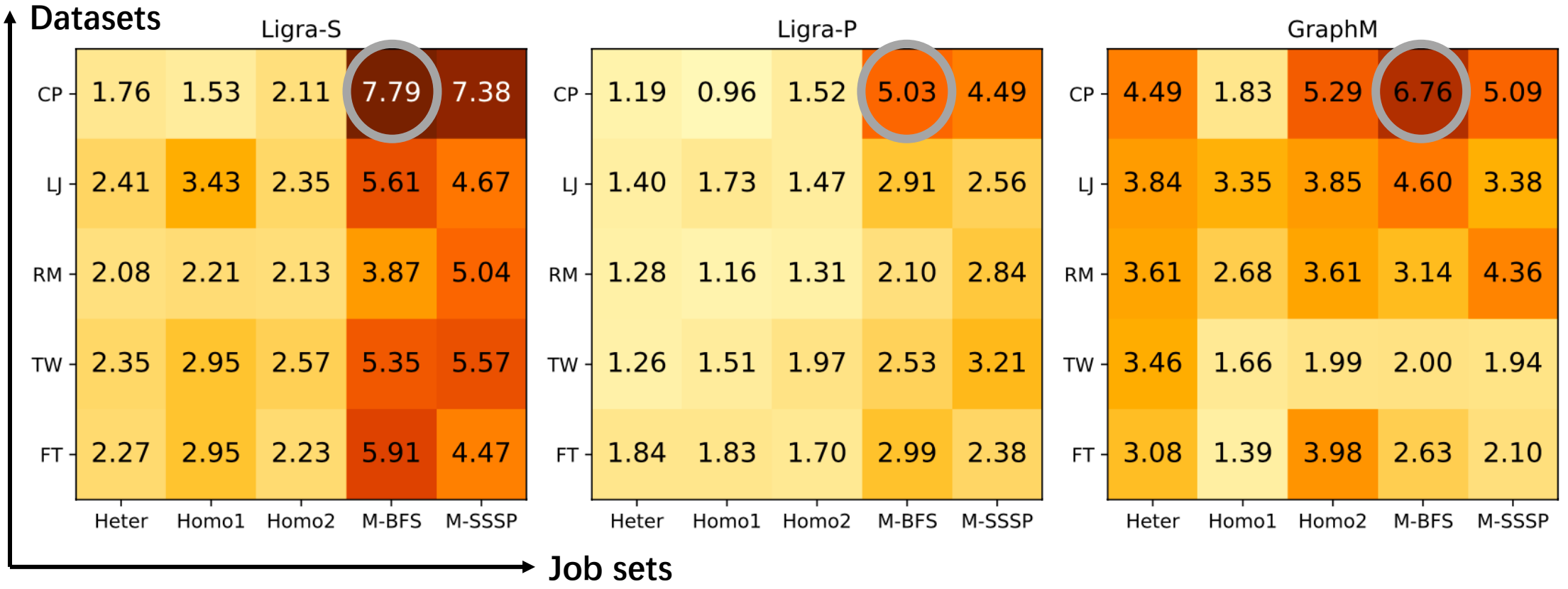
Abbr.	Dataset	Vertices	Edges
CP	cit-Patents [50]	6.0 M	16.5M
LJ	LiveJournal [51]	4.8 M	69 M
RM	rMat24 [52]	33.6 M	168 M
TW	Twitter [53]	41.7 M	1.4 B
FT	Friendster [54]	124 M	1.8 B

Experimental Setup

- Applications
  - Breadth-First Search (BFS)
  - Connected Components (CC)
  - PageRank Delta (PR)
  - BellmanFord (SSSP)
- Machines & Environment
  - 2*Intel Xeon Gold 5118 CPU (48 cores)
  - 768 GB memory
  - 2 nodes for distributed experiment
  - Cilk Plus (shared-mem) / OpenMPI (dist)
- Job sets (8 jobs in a batch submitted simultaneously)

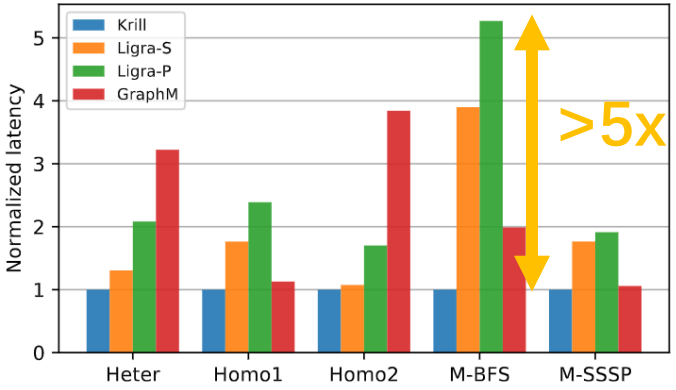
Job set	Algorithms	Description
Heter	$\{BFS, CC, PR, SSSP\} \times 2$	Heterogeneous
Homo1	$\{BFS, CC\} \times 4$	Homogeneous
Homo2	$\{PR, SSSP\} \times 4$	Homogeneous
M-BFS	$\{BFS\} \times 8$	Multiple same jobs
M-SSSP	$\{SSSP\} \times 8$	Multiple same jobs

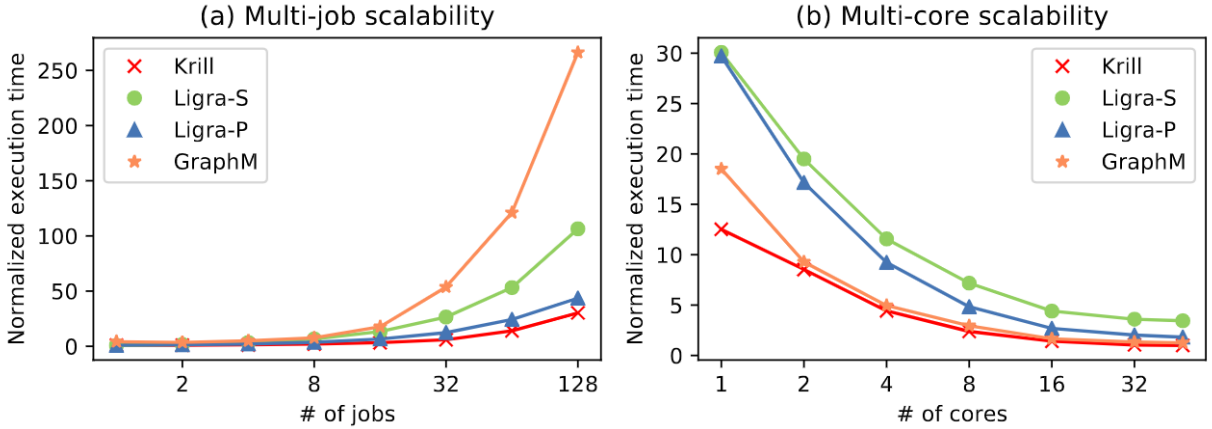
Experimental Results



- Up to 7.79x speedup compared with Ligra, and 6.67x compared with GraphM
- Preference to heterogeneous job sets

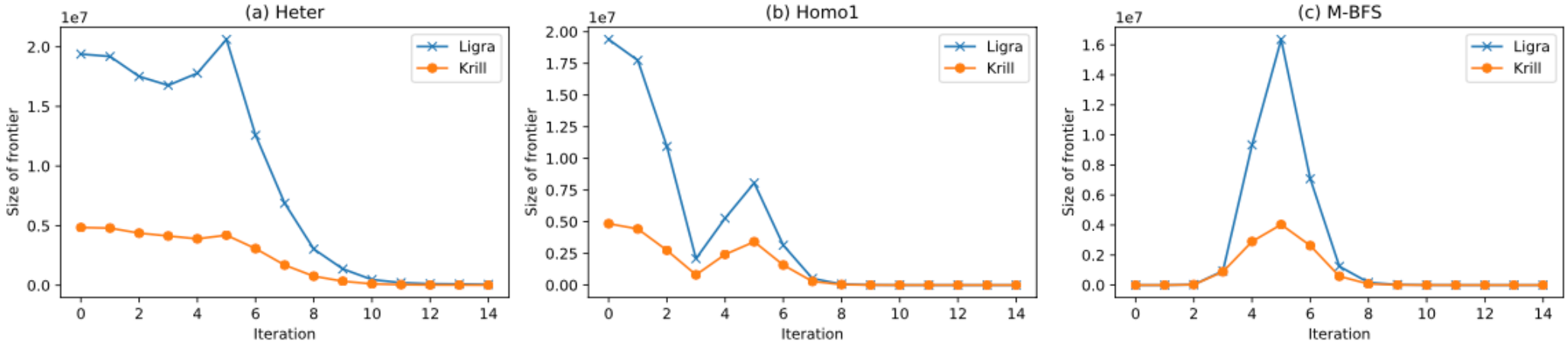
Experimental Results





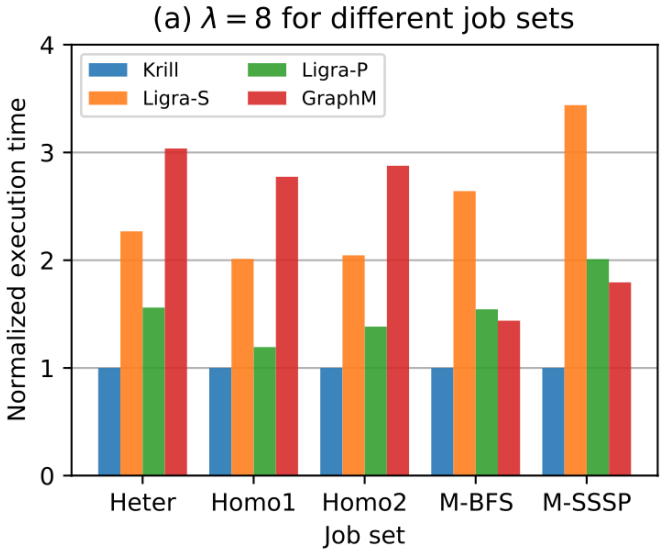
Low response latency

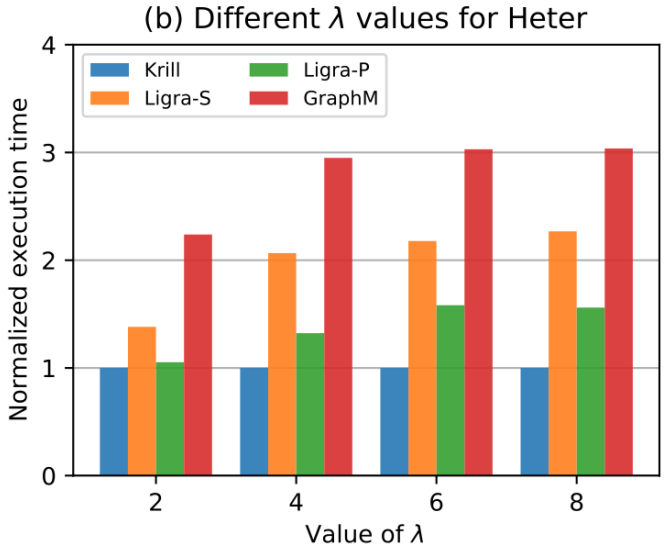
High scalability



Minimum frontier set, reducing graph structure access

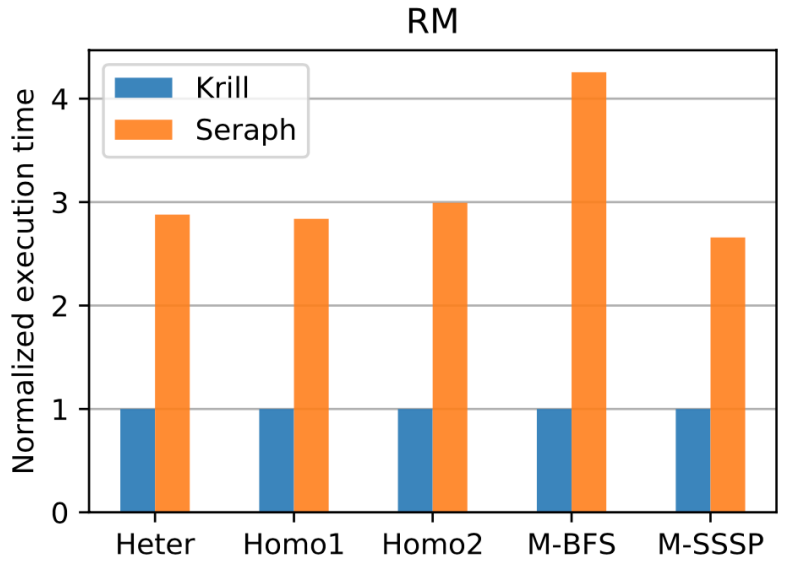
Experimental Results





Dynamic graph job submission

> 3x speedup



Distributed Environment

Max 4.25x speedup

Conclusion

- **Krill**: Concurrent graph processing system
  - High performance, end-to-end, full-stack solution
- **Compiler**: Property buffers
  - Automatic property management
- **Runtime**: Graph kernel fusion
  - Optimal structure access
- Reduce execution time, memory access, respond latency, etc.

Thanks & QA

Krill: A Compiler and Runtime System for Concurrent Graph Processing

Hongzheng Chen, Minghua Shen, Nong Xiao, Yutong Lu

https://github.com/chhzh123/Krill

Artifact Available | Artifact Functional | Results Reproduced