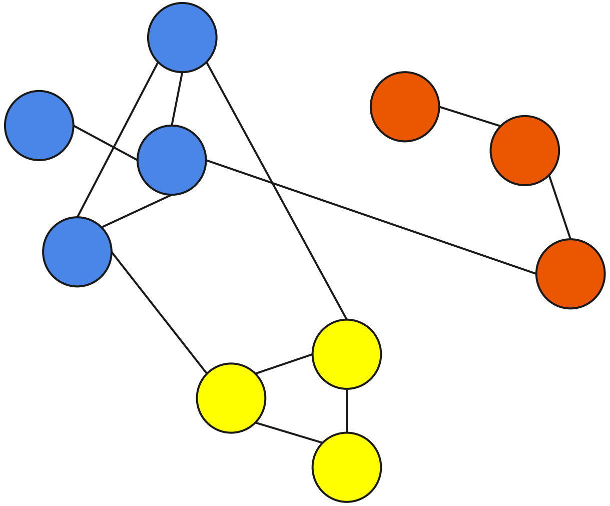
Ligra: A Lightweight Graph Processing Framework for Shared Memory

Paper authors: Julian Shun, Guy Blelloch (Carnegie Mellon University)
Presenter: Mihai-Ionut Enache

25 October 2021

Motivation - Why Study Graphs?

- Many applications: social networks, Web graph, medicine
- Types of problems:
  - Shortest path
  - Clustering (e.g. community recovery)
  - Recommendation engines
  - Scientific computations
  - others



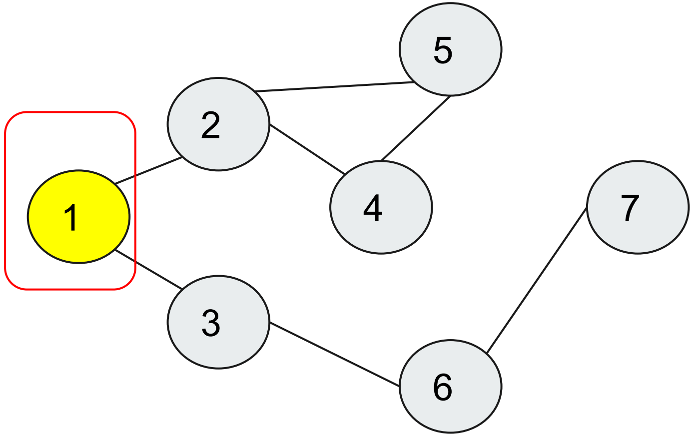
Shared Memory vs. Distributed Systems

- In the past:
  - Memory scarce, few cores available; hard to handle large graphs
  - Most of the frameworks designed to run on distributed systems
- Today:
  - Single multicore commodity computers can have TBs of memory
  - Can accommodate graphs of billions of edges
- Why shared memory?
  - More efficient (per dollar / core / joule)
  - Low communication costs ⇒better performance
  - Simplicity: easier to write algorithms for shared memory
  - More reliable: shared memories can run months / years without failure

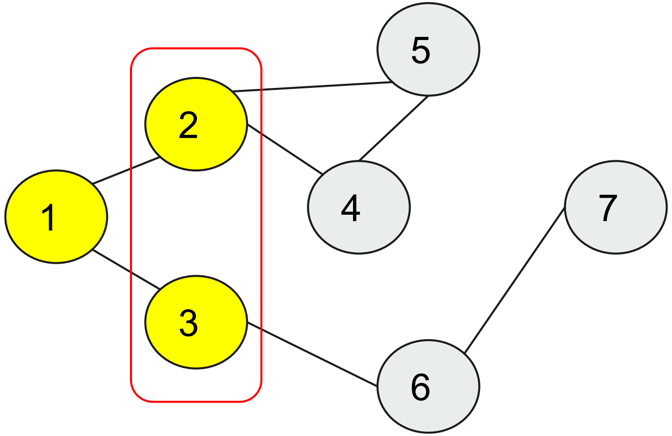
Ligra - Preview

- Lightweight
  - Interface: only a few functions
  - Implementation: simple and fast
- 2 datatypes: one for graph G = (V, E) and one for subsets of V (`VertexSubset`)
- 2 essential functions:
  - **`VertexMap`** (maps over V or subsets of V)
  - **`EdgeMap`**
  - Useful in graph traversal algorithms
- **`Compare-and-swap (CAS)`**: atomic instruction for conditional swapping

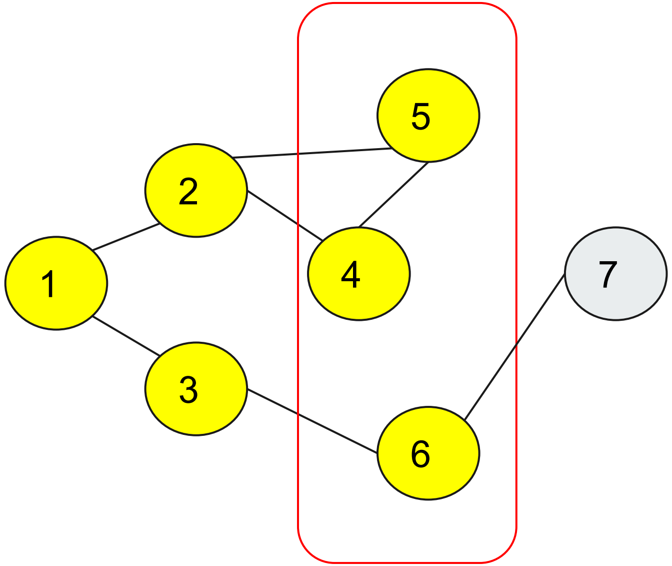
Application: Breadth-first Search



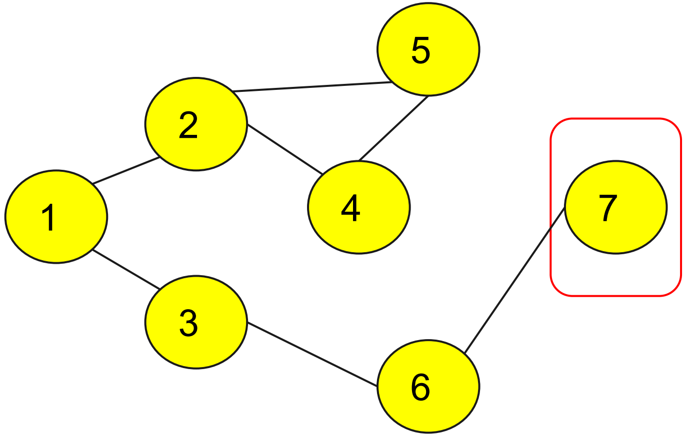
Application: Breadth-first Search



Application: Breadth-first Search



Application: Breadth-first Search



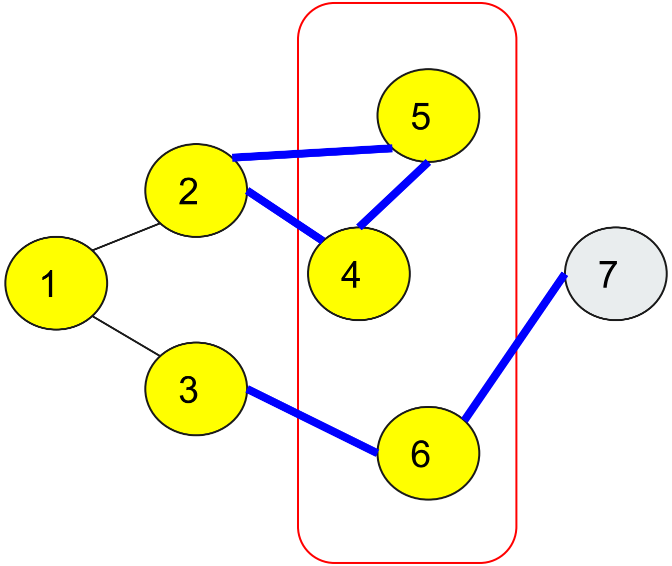
BFS in Ligra

```
Parents = {−1, . . . , −1}                          ▷ initialized to all -1's

procedure UPDATE(s, d)
    return (CAS(&Parents[d], −1 , s ))

procedure COND(i)
    return (Parents[i] == −1)

procedure BFS(G, r)                                  ▷ r is the root
    Parents[r] = r
    Frontier = {r}          ▷ vertexSubset initialized to contain only r
    while (SIZE(Frontier) ≠ 0) do
        Frontier = EDGEMAP(G, Frontier, UPDATE, COND)
```

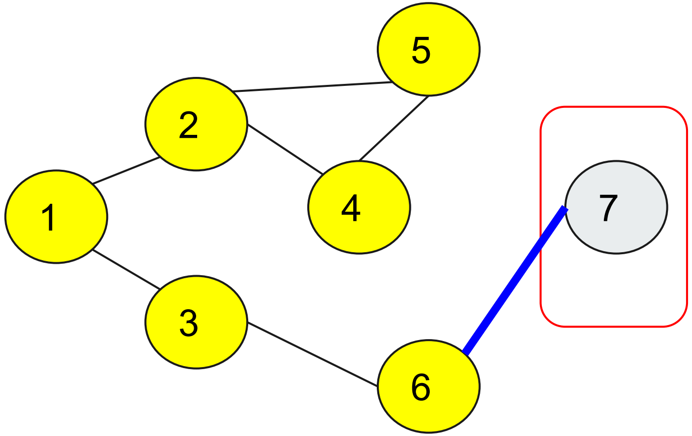
Edge Processing

- Interface allows processing edges in different orders
  - Ligra is edge-oriented
  - Previous systems mostly vertex-oriented
- 3 ways to process edges:
  - **Sparse representation**: iterate over the active source vertices and check (target of) out-edges
  - **Dense representation**: iterate over the destination vertices and check (source of) in-edges
  - Flat map: check all edges

Application: BFS (sparse representation)



Application: BFS (dense representation)



Sparse or Dense Representation?

- Idea: use a hybrid approach
  - Choose based on the size of the frontier and the number of out-edges
  - If larger than a fixed threshold, use dense, otherwise sparse
- Inspired from previous work of an efficient BFS implementation
  - Ligra generalizes the idea

Interface

$$\textsc{EdgeMap}(G : \textit{graph},\ U : \textit{vertexSubset},\ F : (\textit{vertex} \times \textit{vertex}) \mapsto \textit{bool},\ C : \textit{vertex} \mapsto \textit{bool}) : \textit{vertexSubset}.$$

- Apply F on all edges (s, t) s.t. s ∈ U and C(t) hold
- F can run in parallel
- User’s responsibility for parallel correctness
- F can have side effects
- For weighted graphs F takes an additional argument
- C is optional - useful for algorithms when data needs to be updated only once (BFS)

$$\textsc{VertexMap}(U : \textit{vertexSubset},\ F : \textit{vertex} \mapsto \textit{bool}) : \textit{vertexSubset}.$$

- F can run in parallel

EdgeMap - Implementation

Algorithm 1 EDGEMAP

procedure EDGEMAP(G, U, F, C)
  if ($|U|$ + sum of out-degrees of U > threshold) **then**
    return EDGEMAPDENSE(G, U, F, C)
  else return EDGEMAPSPARSE(G, U, F, C)

Implementation

Algorithm 1 EDGEMAP

```
procedure EDGEMAP(G, U, F, C)
    if (|U| + sum of out-degrees of U > threshold) then
        return EDGEMAPDENSE(G, U, F, C)
    else return EDGEMAPSPARSE(G, U, F, C)
```

Algorithm 3 EDGEMAPDENSE

```
procedure EDGEMAPDENSE(G, U, F, C)
    Out = {}
    parfor i ∈ {0, . . . , |V| − 1} do
        if (C(i) == 1) then
            for ngh ∈ N⁻(i) do
                if (ngh ∈ U and F(ngh, i) == 1) then
                    Add i to Out
                if (C(i) == 0) then break
    return Out
```

Implementation

Algorithm 1 EDGEMAP

```
procedure EDGEMAP(G, U, F, C)
    if (|U| + sum of out-degrees of U > threshold) then
        return EDGEMAPDENSE(G, U, F, C)
    else return EDGEMAPSPARSE(G, U, F, C)
```

In parallel

Algorithm 3 EDGEMAPDENSE

```
procedure EDGEMAPDENSE(G, U, F, C)
    Out = {}
    parfor i ∈ {0, . . . , |V| − 1} do
        if (C(i) == 1) then
            for ngh ∈ N⁻(i) do
                if (ngh ∈ U and F(ngh, i) == 1) then
                    Add i to Out
                if (C(i) == 0) then break
    return Out
```

Implementation

Algorithm 1 EDGEMAP

```
procedure EDGEMAP(G, U, F, C)
    if (|U| + sum of out-degrees of U > threshold) then
        return EDGEMAPDENSE(G, U, F, C)
    else return EDGEMAPSPARSE(G, U, F, C)
```

Algorithm 3 EDGEMAPDENSE

```
procedure EDGEMAPDENSE(G, U, F, C)
    Out = {}
    parfor i ∈ {0, . . . , |V| − 1} do
        if (C(i) == 1) then
            for ngh ∈ N⁻(i) do
                if (ngh ∈ U and F(ngh, i) == 1) then
                    Add i to Out
                if (C(i) == 0) then break
    return Out
```

In parallel

Sequentially

Implementation

Algorithm 1 EDGEMAP

```
procedure EDGEMAP(G, U, F, C)
    if (|U| + sum of out-degrees of U > threshold) then
        return EDGEMAPDENSE(G, U, F, C)
    else return EDGEMAPSPARSE(G, U, F, C)
```

Algorithm 2 EDGEMAPSPARSE

```
procedure EDGEMAPSPARSE(G, U, F, C)
    Out = {}
    parfor each v ∈ U do
        parfor ngh ∈ N+(v) do
            if (C(ngh) == 1 and F(v, ngh) == 1) then
                Add ngh to Out
    Remove duplicates from Out
    return Out
```

Algorithm 3 EDGEMAPDENSE

```
procedure EDGEMAPDENSE(G, U, F, C)
    Out = {}
    parfor i ∈ {0, . . . , |V| − 1} do
        if (C(i) == 1) then
            for ngh ∈ N−(i) do
                if (ngh ∈ U and F(ngh, i) == 1) then
                    Add i to Out
                if (C(i) == 0) then break
    return Out
```

Implementation

Algorithm 1 EDGEMAP

```
procedure EDGEMAP(G, U, F, C)
    if (|U| + sum of out-degrees of U > threshold) then
        return EDGEMAPDENSE(G, U, F, C)
    else return EDGEMAPSPARSE(G, U, F, C)
```

Algorithm 2 EDGEMAPSPARSE

```
procedure EDGEMAPSPARSE(G, U, F, C)
    Out = {}
    parfor each v ∈ U do                 } In parallel
        parfor ngh ∈ N+(v) do            }
            if (C(ngh) == 1 and F(v, ngh) == 1) then
                Add ngh to Out
    Remove duplicates from Out
    return Out
```

Algorithm 3 EDGEMAPDENSE

```
procedure EDGEMAPDENSE(G, U, F, C)
    Out = {}
    parfor i ∈ {0, ..., |V| − 1} do
        if (C(i) == 1) then
            for ngh ∈ N−(i) do
                if (ngh ∈ U and F(ngh, i) == 1) then
                    Add i to Out
                if (C(i) == 0) then break
    return Out
```

VertexMap - Implementation

Algorithm 4 VERTEXMAP

```
procedure VERTEXMAP(U, F)
    Out = {}
    parfor u ∈ U do
        if (F(u) == 1) then Add u to Out
    return Out
```

Optimizations

- F in EdgeMapDense is applied sequentially ⇒ doesn't need atomicity
    - Optimized version of EdgeMap: accepts 2 versions of F
    - Authors found this to be slightly faster for some applications
- Users can set a different threshold for EdgeMapSparse vs EdgeMapDense
    - Default is |E| / 20
- Inner-loop of EdgeMapDense can also run in parallel
    - User needs to give up the "break" option to enable this

Algorithm 3 EDGEMAPDENSE

```
procedure EDGEMAPDENSE(G, U, F, C)
    Out = {}
    parfor i ∈ {0, . . . , |V| − 1} do
        if (C(i) == 1) then
            for ngh ∈ N⁻(i) do
                if (ngh ∈ U and F(ngh, i) == 1) then
                    Add i to Out
                if (C(i) == 0) then break
    return Out
```

Applications

1. BFS
2. Betweenness Centrality
3. Graph Radii Estimation and Multiple BFS
4. Connected Components
5. PageRank + PageRank-Delta
6. Bellman-Ford Shortest Paths

Experiments

Input	Num. Vertices	Num. Directed Edges
3D-grid	10^7	6×10^7
random-local	10^7	9.8×10^7
rMat24	1.68×10^7	9.9×10^7
rMat27	1.34×10^8	2.12×10^9
Twitter	4.17×10^7	1.47×10^9
Yahoo*	1.4×10^9	12.9×10^9

Table 1. Graph inputs. *The original asymmetric graph has 6.6×10^9 edges.

Experiments (continued)

Application	3D-grid			random-local			rMat24			rMat27			Twitter			Yahoo		
	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)	(1)	(40h)	(SU)
Breadth-First Search	2.9	0.28	10.4	2.11	0.073	28.9	2.83	0.104	27.2	11.8	0.423	27.9	6.92	0.321	21.6	173	8.58	20.2
Betweenness Centrality	9.15	0.765	12.0	8.53	0.265	32.2	11.3	0.37	30.5	113	4.07	27.8	47.8	2.64	18.1	634	23.1	27.4
Graph Radii	351	10.0	35.1	25.6	0.734	34.9	39.7	1.21	32.8	337	12.0	28.1	171	7.39	23.1	1280	39.6	32.3
Connected Components	51.5	1.71	30.1	14.8	0.399	37.1	14.1	0.527	26.8	204	10.2	20.0	78.7	3.86	20.4	609	29.7	20.5
PageRank (1 iteration)	4.29	0.145	29.6	6.55	0.224	29.2	8.93	0.25	35.7	243	6.13	39.6	72.9	2.91	25.1	465	15.2	30.6
Bellman-Ford	63.4	2.39	26.5	18.8	0.677	27.8	17.8	0.694	25.6	116	4.03	28.8	75.1	2.66	28.2	255	14.2	18.0

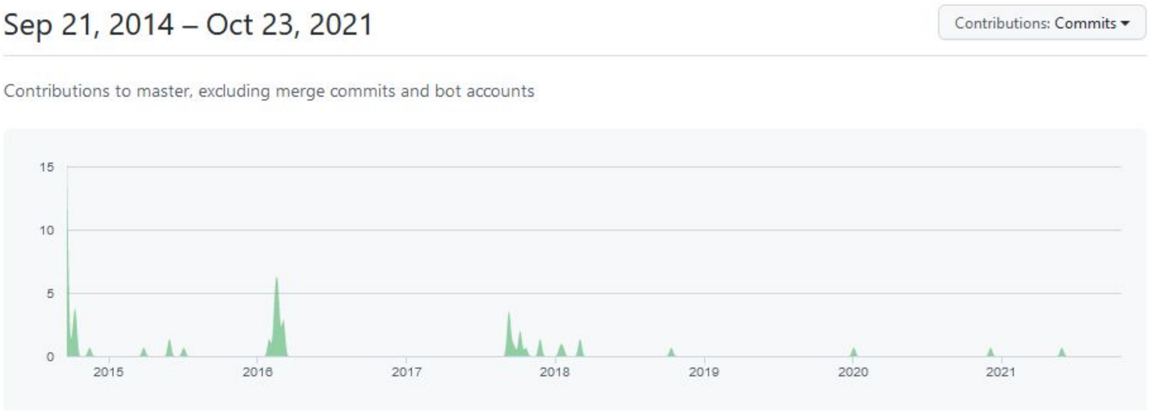
Table 2. Running times (in seconds) of algorithms over various inputs on a 40-core machine (with hyper-threading). (SU) indicates the speedup of the application (single-thread time divided by 40-core time).

Comparison to other frameworks

- Related frameworks: Pregel, KDT, Pegasus, PowerGraph
- **BFS**: **10-28 speedup** + almost as efficient as Beam's highly optimized BFS
- **Betweenness centrality**: **12-32 speedup**
- **Graph radii estimation**: **23-35 speedup**
- **Connected components: 20-37 speedup**
- **PageRank**: **29-39 speedup** for a single iteration
- **Bellman-Ford: 18-28 speedup**

Subsequent work

- Ligra+ - framework for processing compressed graphs (half the space of uncompressed graphs)
- Hygra - framework for hypergraphs (hyperedge = edge with arbitrary number of vertices)



(source: https://github.com/jshun/ligra/graphs/contributors)

Summary

- Ligra is a graph processing framework targeting a class of parallel algorithms
- It comes with a lightweight interface and implementation
- Experimental evaluation shows it performs better than existing work and almost as good as highly optimized code
- Limitation: no support for algorithms that need to modify the input graph