Reviewing the Ligra single-node graph processing framework

Thomas Parks

October 24, 2017

U. of Cam

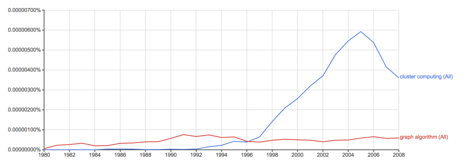
Introduction

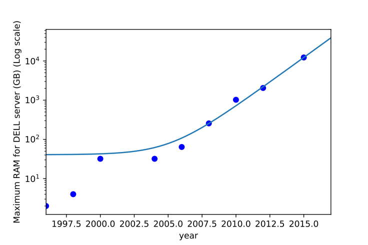
Ligra: A Lightweight Graph Processing Framework for Shared Memory

Authors: Julian Shun and Guy E. Blelloch, CMU

A reaction to the availability of large single nodes.

The shift of computing





Interest in processing graph data has been relatively constant over time, whereas cluster computing fluctuates in the published literature.

The RAM capacity for a single server has grown exponentially, with a knee approximately where the use of clusters drops off

Ligra single node graph computations

API inspired by Hybrid BFS[1].

Aims for every high efficiency by using CAS[2]

Outperforms Pregel on a per core and a absolute basis[3].

Also claims superior performance per dollar and Joule[4].

---

[1] Beamer, Asanovic, et al., *Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500*.
[2] Schweizer, Besta, and Hoefler, "Evaluating the Cost of Atomic Operations on Modern Architectures".
[3] This was not throughly explored in the paper.
[4] This was not mentioned again after claiming improvements in the abstract.

API

Ligra API and motivating example

```
parents = [-1, ..., -1] ;        // The parent of every node
UPDATE s, d
    return CAS(parents[d], -1, s);
COND i
    return parents[i] == -1;
BFS G, r
    parents[r] := r;
    frontier = r;
    while size(frontier) != 0 do
        // For every vertex in the frontier,
           UPDATE all neighbouring j if COND. Add
           to returned set if UPDATE(i, j).
        frontier := EDGEMAP(G, frontier, UPDATE, COND);
    end
```

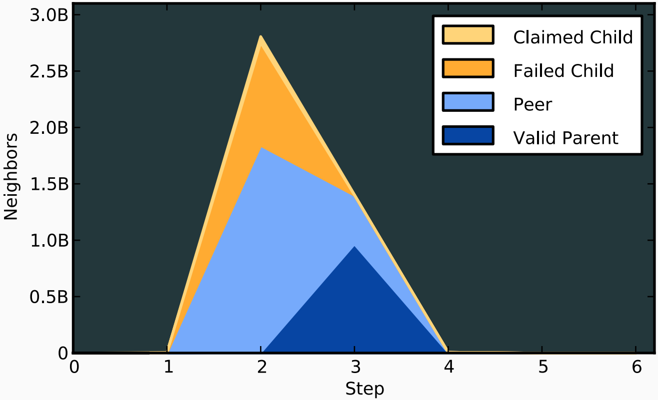
EDGEMAP working outwards

Semantics allow for multiple implementations with different performance.

```
EDGEMAP_SPARSE G, U, F, C
    result = {};
    /* both loops fully parallel                         */
    foreach v in U do
        foreach v2 in out_neighbours(v) do
            if C(v2) and F(v, v2) // not in the BFS tree
            then
                add v2 to result;
            end
        end
    end
    return result;
```

EDGEMAP working outwards

Semantics allow for multiple implementations with different performance.



5

[5] Beamer, Asanović, and Patterson, "Direction-optimizing Breadth-first Search".

EDGEMAP working over all elements

```
EDGEMAP_DENSE G, U, F, C
    result = {};
    /* first loop parallel                                  */
    foreach i in [0, ..., |V(G)|] do
        if C(i) // not in the BFS tree
        then
            foreach v in in_neighbours(i) do
                if v ∈ U and F(v, i) then add i to result;
                if not c(i) then break;
            end
        end
    end
    return result;
```

```
VERTEXMAP U, F
    result = {};
    /* parallel loop                                    */
    foreach u ∈ U do
        if F(u) then add u to result;
    end
    return result;
```

Performance measurements

Ligra scaling

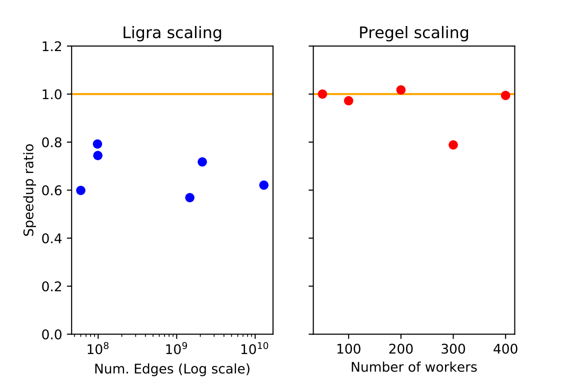


Table 1: 1B vertex binary tree shortest path

Pregel	20 seconds	300 Nodes
Ligra	2 seconds	1 Node

Navigating the maze of Graphs

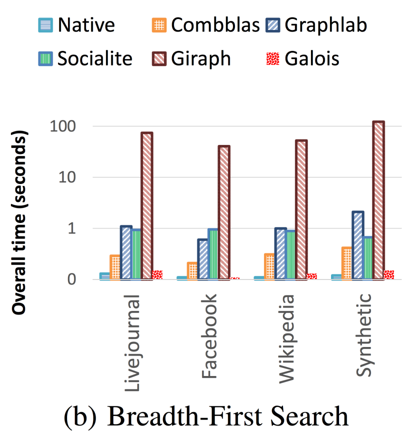


Figure 1: The real performance of algorithms can be hard to find.[6]

[6] Satish et al., "Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets".

Navigating the maze of Graphs

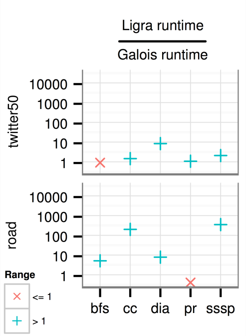


Figure 2: Galois can implement Ligra simply.[7]

[7] Nguyen, Lenharth, and Pingali, “A Lightweight Infrastructure for Graph Analytics”.

Questions?

Nice algo bits.

Graph diameter estimation.

Algorithm 7 Radii Estimation

```
Visited = {0, ..., 0}                                   ▷ initialized to all 0
NextVisited = {0, ..., 0}                               ▷ initialized to all 0
Radii = {∞, ..., ∞}                                     ▷ initialized to all ∞
round = 0

procedure RADIIUPDATE(s, d)
    if (Visited[d] ≠ Visited[s]) then
        ATOMICOR(&NextVisited[d], Visited[d] | Visited[s])
        oldRadii = Radii[d]
        if (Radii[d] ≠ round) then
            return CAS(&Radii[d], oldRadii, round)
    return 0

procedure ORCOPY(i)
    NextVisited[i] = NextVisited[i] | Visited[i]
    return 1

procedure RADII(G)
    Sample K vertices and for each one set a unique bit in Visited to 1
    Initialize Frontier to contain the K sampled vertices
    Set the Radii entries of the sampled vertices to 0
    while (SIZE(Frontier) ≠ 0) do
        round = round + 1
        Frontier = EDGEMAP(G, Frontier, RADIIUPDATE, C_true)
        Frontier = VERTEXMAP(Frontier, ORCOPY)
        SWAP(Visited, NextVisited)                      ▷ switch roles of bit-vectors
    return Radii
```

Associate a bit vector with each vertex for all BFS searches, and bitwise OR the current vertex vector with neighbours. Vertices that change are on the new multiBFS frontier. Store the iteration number of the last time a vertex changed it's vector. This is a lower bound on centrality of that vertex, and *max*(*centrality*) is the diameter.