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.
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.
API inspired by Hybrid BFS\(^1\).

Aims for every high efficiency by using CAS\(^2\).

Outperforms Pregel on a per core and an 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 thoroughly 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.

Figure 3 shows a breakdown of the result of each edge check for each step during a conventional parallel queue-based top-down BFS traversal on a Kronecker-generated synthetic graph (used for the Graph500 benchmark [15]). The middle steps (2 and 3) consume the vast majority of the runtime, which is unsurprising since the frontier is then at its largest size, requiring many more edges to be examined. During these steps, there are a great number of wasted attempts to become the parent of a neighbor. Failures occur when the neighbor has already been visited, and these can be broken down into three different categories based on their depth relative to the candidate parent:

- **Valid parent** is any neighbor at depth $d - 1$ over a tested depth $d$.
- **Peer** is any neighbor at the same depth.
- **Failed child** is any neighbor at depth $d + 1$ over a tested depth $d$, but at the time of examination it has already been claimed by another vertex at depth $d$.

Figure 3 shows how most edge checks do fail and represent redundant work, since a vertex in a correct BFS tree only needs one parent.

Implementations of this same basic algorithm can vary in a number of performance-impacting ways, including: data structures, traversal order, parallel work allocation, partitioning, synchronization or update procedure. The process of checking if neighbors have been visited can result in many costly random accesses. An effective optimization for shared-memory machines with large last-level caches is to use a bitmap to mark nodes that have already been visited [1]. The bitmap can often fit in the last-level cache, which prevents many of those random accesses from touching off-chip DRAM. These optimizations speed up the edge checks but do not reduce the number of checks required.

The theoretical minimum for the number of edges that need to be examined in the best case is the number of vertices in the BFS tree minus one, since that is how many edges are required to connect it. For the example in Fig. 3, only 63,036,116 vertices are in the BFS tree, so at least 63,036,115 edges need to be considered, which is about $\frac{1}{67}$ of all the edge examinations that would happen during a top-down traversal. This factor of 67 is substantially larger than the input degree of 16 for two reasons. First, the input degree is for undirected edges, but during a top-down search each edge will be checked from both endpoints, doubling the number of examinations. Secondly, there are a large number of vertices of zero degree, which reduces the size of the main connected component and also further increases the effective degree of the vertices it contains. There is clearly substantial room for improvement by checking fewer edges, although in the worst case, every edge might still need to be checked.

Figure 4 zooms in on the edge check results of Fig. 3 for the sample search. This progression of neighbor types is typical among the social networks examined. During the first few steps, the percentage of claimed children is high, as the vast majority of the graph is unexplored, enabling most edge checks to succeed. During the next few steps, the percentage of failed children rises, which is unsurprising since the frontier has grown larger, as multiple valid parents are fighting over children. As the frontier reaches its largest size, the percentage of peer edges dominates. Since the frontier is such a large fraction of the graph, many edges must connect vertices within the frontier. As the frontier size rapidly decreases after its apex, the percentage of valid parents rises since such a large fraction of edges were in the previous step's frontier.

---

[^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
**Table 1:** 1B vertex binary tree shortest path

<table>
<thead>
<tr>
<th></th>
<th>Pregel</th>
<th>20 seconds</th>
<th>300 Nodes</th>
<th>Ligra</th>
<th>2 seconds</th>
<th>1 Node</th>
</tr>
</thead>
</table>
Navigating the maze of Graphs

Figure 1: The real performance of algorithms can be hard to find.\textsuperscript{6}

\textsuperscript{6}Satish et al., “Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets”.
Figure 2: Galois can implement Ligra simply.\textsuperscript{7}

\textsuperscript{7}Nguyen, Lenharth, and Pingali, “A Lightweight Infrastructure for Graph Analytics”.
Questions?
Graph diameter estimation.

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 \( \max(\text{centrality}) \) is the diameter.