Ligra: A Lightweight Graph Processing Framework for Shared Memory

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Performance

Keywords Shared Memory, Graph Algorithms, Parallel Programming

Stefan Milosevic (sm2731)

What is Ligra?

- Ligra is a lightweight and efficient framework for graph processing in shared memory. It was designed to make it easier for developers to implement graph algorithms that can run efficiently on multi-core machines
- Well suited for graph traversal problems

Implementations

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
```

Algorithm 4 VERTEXMAP

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

Application (Breadth-First Search)

```
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)
```

Figure 1. Pseudocode for Breadth-First Search in our framework. The compare-and-swap function CAS(*loc*,*oldV*,*newV*) atomically checks if the value at location *loc* is equal to *oldV* and if so it updates *loc* with *newV* and returns *true*. Otherwise it leaves *loc* unmodified and returns *false*.

Application (Betweenness Centrality)

Algorithm 6 Betweenness Centrality

```
NumPaths = {0, ..., 0}                                  ▷ initialized to all 0
Visited = {0, ..., 0}                                   ▷ initialized to all 0
NumPaths[r] = 1
Visited[r] = 1
currLevel = 0
Levels = []
Dependencies = {0.0, ..., 0.0}                          ▷ initialized to all 0.0

procedure VISIT(i)
    Visited[i] = 1
    return 1

procedure PATHSUPDATE(s, d)
    repeat
        oldV = NumPaths[d]
        newV = oldV + NumPaths[s]
    until (CAS(&NumPaths[d], oldV, newV) == 1)
    return (oldV == 0)

procedure DEPUPDATE(s, d)
    repeat
        oldV = Dependencies[d]
        newV = oldV + (NumPaths[d] / NumPaths[s]) × (1 + Dependencies[s])
    until (CAS(&Dependencies[d], oldV, newV) == 1)
    return (oldV == 0.0)

procedure COND(i)
    return (Visited[i] == 0)

procedure BC(G, r)
    Frontier = {r}              ▷ vertexSubset initialized to contain only r
    while (SIZE(Frontier) ≠ 0) do                       ▷ Phase 1
        Frontier = EDGEMAP(G, Frontier, PATHSUPDATE, COND)
        Levels[currLevel] = Frontier
        Frontier = VERTEXMAP(Frontier, VISIT)
        currLevel = currLevel + 1

    Visited = {0, ..., 0}                               ▷ reinitialize to all 0
    currLevel = currLevel − 1
    TRANSPOSE(G)                                        ▷ transpose graph

    while (currLevel ≥ 0) do                            ▷ Phase 2
        Frontier = Levels[currLevel]
        VERTEXMAP(Frontier, VISIT)
        EDGEMAP(G, Frontier, DEPUPDATE, COND)
        currLevel = currLevel − 1
    return Dependencies
```

Application (Radii 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
```

Application (Connected Components)

Algorithm 8 Connected Components

```
IDs = {0, ..., |V| - 1}                ▷ initialized such that IDs[i] = i
prevIDs = {0, ..., |V| - 1}            ▷ initialized such that prevIDs[i] = i

procedure CCUpdate(s, d)
    origID = IDs[d]
    if (writeMin(&IDs[d], IDs[s])) then
        return (origID == prevIDs[d])
    return 0

procedure Copy(i)
    prevIDs[i] = IDs[i]
    return 1

procedure CC(G)
    Frontier = {0, ..., |V| - 1}       ▷ vertexSubset initialized to V
    while (size(Frontier) ≠ 0) do
        Frontier = vertexMap(Frontier, Copy)
        Frontier = edgeMap(G, Frontier, CCUpdate, C_true)
    return IDs
```

Application (PageRank)

Algorithm 9 PageRank

$p_{curr} = \{\frac{1}{|V|}, \ldots, \frac{1}{|V|}\}$ ▷ initialized to all $\frac{1}{|V|}$
$p_{next} = \{0.0, \ldots, 0.0\}$ ▷ initialized to all 0.0
diff $= \{\}$ ▷ array to store differences

procedure PRUPDATE(s, d)
  ATOMICINCREMENT($\&p_{next}[d], \frac{p_{curr}[s]}{deg^{+}(s)}$)
  return 1

procedure PRLOCALCOMPUTE(i)
  $p_{next}[i] = (\gamma \times p_{next}[i]) + \frac{1-\gamma}{|V|}$
  diff$[i] = |p_{next}[i] - p_{curr}[i]|$
  $p_{curr}[i] = 0.0$
  return 1

procedure PAGERANK(G, γ, ϵ)
  Frontier $= \{0, \ldots, |V| - 1\}$ ▷ vertexSubset initialized to V
  error $= \infty$
  while (error $> \epsilon$) **do**
    Frontier = EDGEMAP(G, Frontier, PRUPDATE, C_{true})
    Frontier = VERTEXMAP(Frontier, PRLOCALCOMPUTE)
    error = sum of diff entries
    SWAP(p_{curr}, p_{next})
  return p_{curr}

Application (Bellman-Ford)

Algorithm 10 Bellman-Ford

```
SP = {∞, . . . , ∞}                                   ▷ initialized to all ∞
Visited = {0, . . . , 0}                              ▷ initialized to all 0

procedure BFUPDATE(s, d, edgeWeight)
    if (WRITEMIN(&SP[d], SP[s] + edgeWeight)) then
        return CAS(&Visited[d], 0, 1)
    else return 0

procedure BFRESET(i)
    Visited[i] = 0
    return 1

procedure BELLMAN-FORD(G, r)
    SP[r] = 0
    Frontier = {r}                  ▷ vertexSubset initialized to contain just r
    round = 0
    while (SIZE(Frontier) ≠ 0 and round < |V|) do
        round = round + 1
        Frontier = EDGEMAP(G, Frontier, BF-UPDATE, C_true)
        Frontier = VERTEXMAP(Frontier, BF-RESET)
    if (round == |V|) then return "negative-weight cycle"
    else return SP
```

Running Times

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).

Setup of Experiments

All experiments were performed on:

- 40-core Intel machine (with hyper-threading), 4x2:4GHz Intel

10-core E7-8870 Xeon processors, a 1066MHz bus, (256GB) main memory

- icpc compiler (version 12.1.0) using CilkPlus with the -O3 flag

Conclusion

- **Designed for shared-memory machines.**
- **Implementations using Ligra:**

 - Efficient and scalable

 - Often outperform other graph libraries/systems

- **Potential uses for other algorithms:**

 - Maximum flow

 - Biconnected components

 - Belief propagation

 - Markov clustering

- **Current Limitations:**

 - Doesn't support algorithms that modify the input graph

- **Future Directions:**

 - Extend Ligra to support graph modifications

 - Explore adaptability for GPU systems