ITMO University

Кафедра компьютерных технологий

Grigoriy Tkachenko

Parallel shortest paths algorithms

Saint Petersburg
2015

Table of contents

Chapter 1. Shortest path from one vertex to every other

1.1. Classic sequential Bellman-Ford

Algorithm 1 Classic Bellman-Ford

```
1: procedure ClassicBellmanFord(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     for i = 0 to |G.vertices| − 1 do
5:         for e ∈ G.edges do
6:             dist[e.to] ← min(dist[e.to], dist[e.from] + e.w)
7:     return dist
```

1.2. BFS-like sequential Bellman-Ford

Algorithm 2 BFS-like BellmanFord

```
1: procedure BFSBellmanFord(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     CurrentVertexSet ← {start}          ▷ Set of vertices the distance to which has just been updated
5:     NextVertexSet ← ∅
6:     step ← 0
7:     while step < |G.vertices| and not CurrentVertexSet.empty() do
8:         step ← step + 1
9:         NextVertexSet.clear()
10:        for v ∈ CurrentVertexSet do
11:            for e ∈ G.edgesFrom[v] do                ▷ Outgoing edges of current vertex
12:                if dist[e.to] < dist[e.from] + e.w then
13:                    dist[e.to] ← dist[e.from] + e.w
14:                    NextVertexSet.insert(e.to)
15:        CurrentVertexSet ← NextVertexSet
16:    return dist
```

1.2.1. Parallel Bellman-Ford by edges of current vertex

Idea : use parallel min reduce for incoming edges of current vertex

Algorithm 3 Parallel Bellman-Ford by edges of current vertex

```
1:  procedure BellmanFordPar1(G, start)
2:      dist ← {∞...∞}
3:      dist[start] ← 0
4:      for i = 0 to |G.vertices| − 1 do
5:          changed ← false
6:          for v ∈ G.vertices do
7:              minDist ← min reduce by G.inEdges[v]
8:              if dist[v] > minDist then
9:                  dist[v] ← minDist
10:                 changed ← true
11:         if not changed then
12:             break
13:     return dist
```

1.2.2. Parallel Bellman-Ford by all edges using prefixsum

Idea : use precalculated prefixsum to divide current vertex set for 2 sets, which will be handled by different threads

Algorithm 4 Parallel Bellman-Ford by all edges using prefixsum

```
1: procedure BellmanFordPar2(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     prefsum ← prefix sum by vertices incoming degree
5:     planMap ← BuildPlan(prefsum, 0, |G.vertices|)
6:     for i = 0 to |G.vertices| do
7:         if not ProcessLayer(G, planMap, prefsum, 0, |G.vertices|) then
8:             break
9:     return dist
10:
11: procedure BuildPlan(prefsum, startV, endV) ▷ This function returns a structure which tells us the middle of the segments (by index of start and end vertex)
12:     resultMap ← empty map
13:     edgesNumber ← prefsum[endV] − prefsum[startV]
14:     if edgesNumber < threshold then
15:         midV ← binary search by edges number
16:         resultMap[startV][endV] ← midV
17:         resultMap.addAll( BuildPlan(prefsum, startV, midV))
18:         resultMap.addAll( BuildPlan(prefsum, midV, endV))
19:     return resultMap
20:
21: procedure ProcessLayer(G, planMap, prefsum, startV, endV)
22:     edgesNumber ← prefsum[endV] − prefsum[startV]
23:     if edgesNumber < threshold then
24:         process vertices sequentally
25:     else
26:         midV ← planMap[startV][endV]
27:         ProcessLayer(G, planMap, prefsum, startV, midV)
28:         ProcessLayer(G, planMap, prefsum, midV, endV)
```

1.2.3. Parallel BFS-like Bellman-Ford

Idea : use your PBFS to handle vertex distances

Algorithm 5 Parallel BFS-like Bellman-Ford

```
1: procedure BellmanFordPar3(G, start)
2:     dist ← {∞...∞}
3:     dist[start] ← 0
4:     Frontier ← {G.edgesFrom(start)}
5:     for i = 0 to |G.vertices| do
6:         Frontier ← HandleFrontier(Frontier)        ▷ relax edges from Frontier and build a new one
7:         if Frontier.empty() then
8:             break
9:     return dist
10:
11: procedure HandleFrontier(Frontier)
12:     recursively divide current frontier, atomically relax edges in frontier and building a new one
13:     return NewFrontier
```

1.2.4. Algo comparison

At the first sight it may seems that Algorithm 3 has only disadvantages. The main problem is that it has bad parallelisation ability compared to other two algorithms. But sometimes it works better even on 40-core machine because of one useful property. Let's consider a graph where all the edges have the from $i \to j$ where $i < j$. Once the iteration number I has passed all the vertices till I have correct target distance. It' easy to prove using mathematic induction. So it means that we have to perform only 2(!!!) loops in that case. One for calculating distance and one for understanding that nothing will change anymore. And let's assume that we have a dense graph. In that circumstances Algorithm 4 will suffer from great parallelism (the number of iterations of the main loop will increase significantly) and Algorithm 5 will have to handle the large set of vertices (because graph is dense) during the iterations.

But anyway it's easy to see that in the most cases Algorithm 4 and Algorithm 5 will beat Algorithm 3. Let's compare them. As I said before Algorithm 5 will be not good enough when we're considering dense graphs, because of big size of queue. So the main recommendation of when to use these

approaches is to realise if the graph is dense or sparse. In the first case you have to use Algorithm 4, otherwise Algorithm 5.

1.2.5. Testing

Now we'll prove our assumptions on practice. I've implemented all the algorithms and compared them. Description of input graphs is presented in the Table 1.1. The results are presented in the Table 1.2

Name	Vertices	Edges
CompleteTS sign(-)	7071	24995985
Complete sign	3162	9995082
BalancedTree fraction	8388607	8388608
SquareGrid sign	2499561	4999122
RandomSparse fraction(0.5) sign	2500000	25000000
RandomSparse fraction(0.96) sign(+)	2500000	25000000
RandomDense fraction(0.5) sign	5000	25000000
RandomDense fraction(0.96) sign(+)	5000	25000000

sign - sign of weights on edges

fraction - fraction of lexicographically sorted edges (edges of type X -> X+i)

TS - exists only Lexicographically Sorted edges (fraction = 1)

expression "*RandomDense fraction(0.5) sign*"means Random Dense graph with specified fraction and any sign of weights

Table 1.1: Input graph description

Algo №	Complete			BalancedTree		SquareGrid		RandomSparse			RandomDense		
	TS-	+	-	0.5	1	+	+-	0.5+	0.5-	0.96+	0.5+	0.5-	0.96+
3	2.43	4.65	nc	116.31	9.04	5.49	13.40	nc	nc	24.35	nc	nc	5.01
4	5.17	0.18	10.84	3.59	3.08	5.92	7.10	2.77	14.68	2.42	0.48	6.38	0.46
5	44.63	0.37	23.55	0.44	0.31	4.42	0.58	0.98	22.59	0.76	0.60	10.25	0.71

Table 1.2: Bellman-Ford algorithms comparison

1.3. Conclusion

You can easily find out from tables that our assumptions were correct. Algorithm 3 works good for dense graph with very high fraction (almost 1), Algorithm 4 is good for dense graphs and for graphs with negative edges, Algorithm 5 is good for sparse graph with positive edges.

Chapter 2. All-pairs shortest distance problem

2.1. Floyd-Warshall algorithm

The most popular algorithm. $O(V^3)$

Algorithm 6 Floyd-Warshall

```
1: procedure FLOYD(G)
2:     dist ← {{∞ ... ∞} ... {∞ ... ∞}}
3:     for e ∈ G.edges do
4:         dist[e.from][e.to] ← e.w
5:
6:     for i = 0 to |G.vertices| do
7:         for u = 0 to |G.vertices| do
8:             for v = 0 to |G.vertices| do
9:                 dist[u][v] ← min(dist[u][v], dist[u][i] + dist[i][v])
10:    return dist
```

2.2. First parallel version

Idea : run parallel Bellman-Ford from every vertex. At the same time use simple scheduling for our computation

Algorithm 7 First parallel version

```
1: procedure ALLPAIRSPAR1(G)
2:     return HANDLEVERTICES(G, 0, |G.vertices|)
3:
4: procedure HANDLEVERTICES(G, startVertex, endVertex)
5:     if endVertex − startVertex < threshold then
6:         distances ← run parallel Bellman-Ford for every vertex from startVertex to endVertex (start -
   inclusively, end - exclusively)
7:         return distances
8:     else
9:         midV ← (startVertex + endVertex)/2
10:        fork2(
               HANDLEVERTICES(G, startV, midV),
               HANDLEVERTICES(G, midV, endV));
```

2.3. Second parallel version

The potential improvement of the idea above was to calculate distance for group of vertices at the same time. How does we achieve it? All we have to do is to create a large graph consisting of V^2 vertices. Every vertex of that graph are of the form (i, j), where i - current vertex id (the same as in the initial graph), j - vertex from which the shortest distance is calculating. And when we're performing Bellman-Ford we have to push all the vertices of form (i, i) to the first Frontier. Afterwards the process would be the same as before. Finally, when we've done with algorithm, the shortest distance d to vertex (i, j) means that shortest distance in the initial graph from the vertex j to vertesx i equals to d.

Unfortunately as we'll see in the following chapters this algorithms occurs to be slower than previous one. However the idea of calculating distances for the group of vertices underlies the following algorithms for social graphs.

2.4. Third parallel version. Algorithm for social graphs

In the following chapter I will describe algorithms to calculate all-pairs distances for social undirected unweighted graphs.

2.4.1. Idea

The algorithm is based on a well-known theory "Six degrees of separation"(`https://en.wikipedia.org/wiki/Six_degrees_of_separation`). According to that the distance beetwen any two vertices in social graph in general is quite small. So the algorithm would use this fact and calculate distances for two sets separately - for a big set of near vertex to some specific base vertex and for remaining vertices. For the big set the

algorithm will use dynamic programming and for the small one it will use previous approaches.

2.4.2. Algorithm

The algorithm consists of three phases

1. Graph analysis and base vertex selection
2. Small set handling
3. Big set handling

First phase is the easiest one. The pseudocode is below (K in pseudocode is the max distance from the base vertex to be handled by it)

Algorithm 8 First phase

```
1: procedure ConstructSets(G)
2:     baseVertex ← vertex with max degree
3:     dist ← run serial bfs from baseVertex
4:     handleByBaseVertexSet ← ∅
5:     parfor i = 0 to |G.vertices| − 1 do
6:         if dist[i] ≤ K then
7:             handleByBaseVertexSet.add(i)
8:     otherVertexSet ← G.vertices \ handleByBaseVertexSet
9:     return baseVertex, handleByBaseVertexSet, otherVertexSet
```

Second phase is essentially the First parallel algorithm (parallel Bellman-Ford from every vertex) but not for the whole set of vertices but for the $otherVertexSet$.

Third phase is based on the dynamic programming. Let's first write down some observations.

Consider some vertex i such that $dist[baseVertex][i] = d$. For which vertices j the statement $dist[i][j] = s$ may turns to be true? It may appear only for vertices j such that $s - d \leq dist[baseVertex][j] \leq s + d$. It means that if we perform a BFS and put every vertex from $handleByBaseVertexSet$ to initial set then during this BFS the vertex j may appear only on layers $[t - K, t + K]$ where $t = dist[baseVertex][j]$. Therefore we may put every vertex to all the

possible layers for that vertex (this amount of layers is not more than $2K + 1$ what is generally very small number). And thus build a Frontier for every layer. Afterwards we'll use the classic technique to handle Frontier in parallel. But for now we haven't touched the idea of dynamic programming. Let's see how it may be useful in our case.

During the computation we are dealing with two dynamics. First is $mask[u][i]$ - the bitset of vertices such that the distance from them to u is equals to i. Second is $calc[u][i]$ - the bitset of vertices such that distance from them to u is less than i. Bitsets only have to support logical operation (or, and, not) and setting/getting the value in specific bit. For every vertex in $handleByBaseVertexSet$ there is a bit that corresponds to the vertex (we have a bijection beetwen vertices in set and bits in bitset). Obvious observation is that for every vertex u we have to store only $2K + 1$ bitsets (follows from the previous paragraph).

Let's see how we calculate dynamics. It's easy to realise these formulas (2.1 for mask and 2.2 for calc).

$$mask[v][i] = \neg calc[v][i-1] \wedge \bigvee_{\exists (u,v) \in E} mask[u][i-1] \tag{2.1}$$

$$calc[v][i] = calc[v][i-1] \vee mask[v][i] \tag{2.2}$$

These calculations present at the pseudocode below. Pay attention that for every vertex we have to store only $2K + 1$ bitsets. But for better understanding we do not deal with such an implementation details and access to j layer of vertex i just by calling $mask[i][j]$. Another notice is that for the calculation of values of layer i we have to handle Frontier from the previous layer $i-1$. That's why we have $layerToCalc$ and $frontierLayer = layerToCalc-1$ in the last loop.

Algorithm 9 Dynamics calculation

```
1: K ← 6                                                          ▷ Max deep to baseVertex...
2:
3: procedure CalculateDistancesForBigSet(G, baseVertex, handleByBaseVertexSet)
4:     maxLayer ← calculate max distance from baseVertex
5:     Frontiers ← {Frontier_0 ... Frontier_{maxLayer+K}}          ▷ Frontier for every BFS layer
6:     VertexSets ← {VertexSet_0 ... VertexSet_{maxLayer+K}}       ▷ VertexSet for every BFS layer
7:     mask ← {{bitVector(0) ... bitVector(0)} ... {bitVector(0) ... bitVector(0)}}
8:     calc ← {{bitVector(0) ... bitVector(0)} ... {bitVector(0) ... bitVector(0)}}
9:
10:    for i = 0 to maxLayer + K do
11:        for j = dist[baseVertex][i] − K to dist[baseVertex][i] + K do
12:            Frontiers[j].pushEdgesOf(i)
13:            VertexSets[j].addVertex(i)
14:
15:    for v ∈ handleByBaseVertexSet do
16:        mask[v][0].setBit(bitNum(v), 1) ▷ Let's assume that we have a function bitNum which by input
   vertex returns the index of corresponding bit in bitsets
17:
18:    for layerToCalc = 1 to maxLayer + K do
19:        frontierLayer ← layerToCalc − 1
20:        ProcessLayerLazy(G, Frontiers[frontierLayer], mask, layerToCalc)
21:        parfor v ∈ VertexSets[layerToCalc] do
22:            calc[v][layerToCalc] ← mask[v][layerToCalc]
23:            if layerToCalc is not first layer for vertex v then
24:                calc[v][layerToCalc − 1] ← ¬calc[v][layerToCalc − 1]
25:                mask[v][layerToCalc] ← mask[v][layerToCalc] ∧ calc[v][layerToCalc − 1]
26:                calc[v][layerToCalc − 1] ← ¬calc[v][layerToCalc − 1]
27:                calc[v][layerToCalc] ← calc[v][layerToCalc] ∨ calc[v][layerToCalc − 1]
```

Pseudocode of Frontier handling:

Algorithm 10 Frontier handling

```
1: procedure ProcessLayerLazy(G, Frontier, mask, layer, dists, baseVertex)
2:     while not Frontier.empty() do
3:         if hasIncomingQuery() then
4:             if Frontier.nbEdges() ≤ cutoff then
5:                 rejectQuery()
6:             else
7:                 NewFrontier ← ∅
8:                 Frontier.split(NewFrontier)
9:                 fork2(
                       ProcessLayerLazy(G, Frontier, mask, layer, dists, baseVertex),
                       ProcessLayerLazy(G, NewFrontier, mask, layer, dists, baseVertex));
10:        Frontier.iterNumber(pollingCutoff, updateFunction(mask, from, to, layer, dists, baseVertex))
11:
12: procedure UpdateFunction(mask, from, to, layer, dists, baseV)
13:     if HaveOnLayer(layer - 1, from, dists, baseV) and HaveOnLayer(layer, to, dists, baseV) then
14:         mask[to][layer] ← mask[to][layer] ∨ mask[from][layer − 1]                    ▷ Атомарно
15:
16: procedure HaveOnLayer(layer, vertex, dists, baseV)
17:     return |layer − dists[baseV][vertex]| ≤ K
```

Finally how we can restore an answer by this dynamic? For every $mask[u][i]$ we'll find all the one-bits. The one-bit in position j says that the distance from the vertex v (with $bitNum(v) = j$) to vertex u is equal to i. That's how we can fill the answer.

The final pseudocode looks like this:

Algorithm 11 Parallel algorithm for social graphs

```
1: procedure AllPairsSocialPar(G)
2:     dist ← {{∞ ... ∞} ... {∞ ... ∞}}
3:     baseVertex, handleByBaseVertexSet, otherVertexSet ← ConstructSets(G)
4:     AllPairsPar1(G, otherVertexSet, dist)
5:     CalculateDistancesForBigSet(G, baseVertex, otherVertexSet, dist)
6:
7:     parfor v = 0 to |G.vertices| do
8:         for j = dist[baseVertex][i] − K to dist[baseVertex][i] + K do
9:             parfor u ∈ handleByBaseVertexSet do
10:                if mask[v][j].getBit(bitNum(u)) = 1 then
11:                    dist[u][v] = j
12:    return dist
```

2.4.3. Algorithms comparison

The last algorithm doesn't have better asymptotic time, but it has several advantages.

- Distance calculation performs faster than before because of bit operations (and, or, not).
- All the bit operations don't require any extra memory allocated. Once we allocated the memory we do all the operations right there.
- We don't have to do job of merging frontier during the calculations. All the frontier have already been built during the initialisation phase.
- All the frontier are big enough what increase their ability to be parallelised.
- All the other phases are good to be parallelised as well.

To prove these advantages on practice I've implemented algorithms and tested them on Twitter subgraph with 81306 vertices and 4841532 edges (undirected; unweighted). Results are in the table 2.1

Algorithm	Twitter graph
First parallel algorithm	427.217
Algorithm for social graphs	210.322

Table 2.1: Algo comparison

As we see we have more than 2x speedup.